Stream Fusion in HOL

Andreas Lochbihler Alexandra Maximova

September 13, 2023

Stream Fusion is a system for removing intermediate list data structures from functional programs, in particular Haskell. This entry adapts stream fusion to Isabelle/HOL and its code generator. We define stream types for finite and possibly infinite lists and stream versions for most of the fusible list functions in the theories *List* and *Coinductive-List*, and prove them correct with respect to the conversion functions between lists and streams. The Stream Fusion transformation itself is implemented as a simproc in the preprocessor of the code generator.

Brian Huffman's AFP entry [3] formalises stream fusion in HOLCF for the domain of lazy lists to prove the GHC compiler rewrite rules correct. In contrast, this work enables Isabelle's code generator to perform stream fusion itself. To that end, it covers both finite and coinductive lists from the HOL library and the Coinductive entry. The fusible list functions require specification and proof principles different from Huffman's.

Contents

1 Stream fusion implementation **4**

2 Stream fusion for finite lists **4**

2.1 The type of generators for finite lists . . . 4
2.2 Conversion to *'a list* . . . 7
2.3 Producers . . . 9
2.3.1 Conversion to streams . . . 9
2.3.2 *replicate* . . . 9
2.3.3 *upt* . . . 10
2.3.4 *upto* . . . 10
2.3.5 [] . . . 11
2.4 Consumers . . . 11
2.4.1 (!) . . . 11
2.4.2 *length* . . . 11
2.4.3 *foldr* . . . 12
2.4.4 *foldl* . . . 12
2.4.5 *fold* . . . 13
2.4.6 *List.null* . . . 13
2.4.7 *hd* . . . 14
2.4.8 *last* . . . 14
2.4.9 *sum-list* . . . 14
2.4.10 *list-all2* . . . 15
2.4.11 *list-all* . . . 15
2.4.12 *ord.lexordp* . . . 16
2.5 Transformers . . . 17
2.5.1 *map* . . . 17
2.5.2 *drop* . . . 18
2.5.3 *dropWhile* . . . 19
2.5.4 *take* . . . 20
2.5.5 *takeWhile* . . . 21
2.5.6 (@) . . . 21
2.5.7 *filter* . . . 22
2.5.8 *zip* . . . 23
2.5.9 *tl* . . . 25
2.5.10 *butlast* . . . 26
2.5.11 *concat* . . . 27
2.5.12 *splice* . . . 28
2.5.13 *list-update* . . . 30
2.5.14 *removeAll* . . . 32
2.5.15 *remove1* . . . 32
2.5.16 (#) . . . 34
2.5.17 *List.maps* . . . 34

3 Stream fusion for coinductive lists **39**
3.1 Conversions to *'a llist* . . . 40
3.1.1 Infinitely many consecutive *Skip*s . . . 40
3.1.2 Finitely many consecutive *Skip*s . . . 41
3.2 Producers . . . 41
3.2.1 Conversion to streams . . . 41
3.2.2 *iterates* . . . 42
3.2.3 *unfold-llist* . . . 42
3.2.4 *inf-llist* . . . 43
3.3 Consumers . . . 43
3.3.1 *lhd* . . . 43
3.3.2 *llength* . . . 43
3.3.3 *lnull* . . . 44
3.3.4 *llist-all2* . . . 45
3.3.5 *lnth* . . . 46
3.3.6 *lprefix* . . . 47
3.4 Transformers . . . 47
3.4.1 *lmap* . . . 47
3.4.2 *ltake* . . . 48
3.4.3 *ldropn* . . . 49
3.4.4 *ldrop* . . . 50
3.4.5 *ltakeWhile* . . . 51
3.4.6 *ldropWhile* . . . 52
3.4.7 *lzip* . . . 52
3.4.8 *lappend* . . . 54
3.4.9 *lfilter* . . . 56
3.4.10 *llist-of* . . . 56

4 Examples and test cases for stream fusion **57**
4.1 Micro-benchmarks from Farmer et al. [2] . . . 57
4.2 Test stream fusion in the code generator . . . 58

1 Stream fusion implementation

theory *Stream-Fusion*
imports
  Main
begin

ML-file ‹*stream-fusion.ML*›

simproc-setup *stream-fusion* (*f x*) = ‹*K Stream-Fusion.fusion-simproc*›
declare [[*simproc del*: *stream-fusion*]]

Install stream fusion as a simproc in the preprocessor for code equations

setup ‹*Code-Preproc.map-pre* (*fn ss => ss addsimprocs* [@{*simproc stream-fusion*}])›

end

2 Stream fusion for finite lists

theory *Stream-Fusion-List*
imports *Stream-Fusion*
begin

lemma *map-option-mono* [*partial-function-mono*]:
  mono-option f ⟹ *mono-option* (λ*x*. *map-option g* (*f x*))
apply (*rule monotoneI*)
apply (*drule* (*1*) *monotoneD*)
apply (*auto simp add*: *flat-ord-def split*: *option.split*)
done

2.1 The type of generators for finite lists

datatype ('*a*, '*s*) *step* = *Done* | *is-Skip*: *Skip* '*s* | *is-Yield*: *Yield* '*a* '*s*

type-synonym ('*a*, '*s*) *raw-generator* = '*s* ⇒ ('*a*,'*s*) *step*

Raw generators may not end in *Done*, but may lead to infinitely many *Yield*s in a row. Such generators cannot be converted to finite lists, because it corresponds to an infinite list. Therefore, we introduce the type of generators that always end in *Done* after finitely many steps.

inductive-set *terminates-on* :: ('*a*, '*s*) *raw-generator* ⇒ '*s set*
  for *g* :: ('*a*, '*s*) *raw-generator*
where
  stop: *g s* = *Done* ⟹ *s* ∈ *terminates-on g*

| *pause*: ⟦ *g s* = *Skip s'*; *s'* ∈ *terminates-on g* ⟧ ⟹ *s* ∈ *terminates-on g*
| *unfold*: ⟦ *g s* = *Yield a s'*; *s'* ∈ *terminates-on g* ⟧ ⟹ *s* ∈ *terminates-on g*

definition *terminates* :: ('*a*, '*s*) *raw-generator* ⇒ *bool*
where *terminates g* ⟷ (*terminates-on g* = *UNIV*)

lemma *terminatesI* [*intro?*]:
(⋀*s*. *s* ∈ *terminates-on g*) ⟹ *terminates g*
by (*auto simp add*: *terminates-def*)

lemma *terminatesD*:
terminates g ⟹ *s* ∈ *terminates-on g*
by (*auto simp add*: *terminates-def*)

lemma *terminates-on-stop*:
terminates-on (λ-. *Done*) = *UNIV*
by (*auto intro*: *terminates-on.stop*)

lemma *wf-terminates*:
assumes *wf R*
and *skip*: ⋀*s s'*. *g s* = *Skip s'* ⟹ (*s'*,*s*) ∈ *R*
and *yield*: ⋀*s s' a*. *g s* = *Yield a s'* ⟹ (*s'*,*s*) ∈ *R*
shows *terminates g*
proof (*rule terminatesI*)
fix *s*
from ‹*wf R*› **show** *s* ∈ *terminates-on g*
proof (*induction rule*: *wf-induct* [*rule-format*, *consumes 1*, *case-names wf*])
case (*wf s*)
show *?case*
proof (*cases g s*)
case (*Skip s'*)
hence (*s'*, *s*) ∈ *R* **by** (*rule skip*)
hence *s'* ∈ *terminates-on g* **by** (*rule wf.IH*)
with *Skip* **show** *?thesis* **by** (*rule terminates-on.pause*)
next
case (*Yield a s'*)
hence (*s'*, *s*) ∈ *R* **by** (*rule yield*)
hence *s'* ∈ *terminates-on g* **by** (*rule wf.IH*)
with *Yield* **show** *?thesis* **by** (*rule terminates-on.unfold*)
qed (*rule terminates-on.stop*)
qed
qed

context fixes *g* :: ('*a*, '*s*) *raw-generator* **begin**

```
partial-function (option) terminates-within :: 's ⇒ nat option where
  terminates-within s = (case g s of
     Done ⇒ Some 0
  | Skip s' ⇒ map-option (λn. n + 1) (terminates-within s')
  | Yield a s' ⇒ map-option (λn. n + 1) (terminates-within s'))

lemma terminates-on-conv-dom-terminates-within:
  terminates-on g = dom terminates-within
proof (rule set-eqI iffI)+
  fix s
  assume s ∈ terminates-on g
  hence ∃ n. terminates-within s = Some n
    by induction (subst terminates-within.simps, simp add: split-beta)+
  then show s ∈ dom terminates-within by blast
next
  fix s
  assume s ∈ dom terminates-within
  then obtain n where terminates-within s = Some n by blast
  then show s ∈ terminates-on g
  proof (induction rule: terminates-within.raw-induct[rotated 1, consumes 1])
    case (1 terminates-within s s')
    show ?case
    proof(cases g s)
      case Done
      thus ?thesis by (simp add: terminates-on.stop)
    next
      case (Skip s')
      hence s' ∈ terminates-on g using 1 by(auto)
      thus ?thesis using ‹g s = Skip s'› by (simp add: terminates-on.pause)
    next
      case (Yield a s')
      hence s' ∈ terminates-on g using 1 by(auto)
      thus ?thesis using ‹g s = Yield a s'› by (auto intro: terminates-on.unfold)
    qed
  qed
qed

end

lemma terminates-wfE:
  assumes terminates g
  obtains R
  where wf R
```

```
    ⋀s s'. (g s = Skip s') ⟹ (s',s) ∈ R
    ⋀s a s'. (g s = Yield a s') ⟹ (s',s) ∈ R
proof −
  let ?R = measure (λs. the (terminates-within g s)) :: ('a × 'a) set
  have wf ?R by simp
  moreover {
    fix s s'
    assume g s = Skip s'
    moreover from assms have s' ∈ terminates-on g by (rule terminatesD)
    then obtain n where terminates-within g s' = Some n
      unfolding terminates-on-conv-dom-terminates-within by (auto)
    ultimately have the (terminates-within g s') < the (terminates-within g s)
      by (simp add: terminates-within.simps)
    hence (s',s) ∈ ?R by (auto)
  } moreover {
    fix s s' a
    assume 2: g s = Yield a s'
    moreover from assms have s' ∈ terminates-on g by (rule terminatesD)
    then obtain n where terminates-within g s' = Some n
      unfolding terminates-on-conv-dom-terminates-within by (auto)
    ultimately have (s',s) ∈ ?R
      by simp (subst terminates-within.simps, simp add: split-beta)
  } ultimately
  show thesis by (rule that)
qed

typedef ('a,'s) generator = {g :: ('a,'s) raw-generator. terminates g}
  morphisms generator Generator
proof
  show (λ-. Done) ∈ ?generator
    by (simp add: terminates-on-stop terminates-def)
qed

setup-lifting type-definition-generator
```

2.2 Conversion to 'a list

```
context fixes g :: ('a, 's) generator begin

function unstream :: 's ⇒ 'a list
where
  unstream s = (case generator g s of
    Done ⇒ []
  | Skip s' ⇒ unstream s'
```

```
  | Yield x s' ⇒ x # unstream s')
by pat-completeness auto
termination
proof −
  have terminates (generator g) using generator[of g] by simp
  thus ?thesis by(rule terminates-wfE)(erule termination)
qed

lemma unstream-simps [simp]:
  generator g s = Done ⟹ unstream s = []
  generator g s = Skip s' ⟹ unstream s = unstream s'
  generator g s = Yield x s' ⟹ unstream s = x # unstream s'
by(simp-all)

declare unstream.simps[simp del]

function force :: 's ⇒ ('a × 's) option
where
  force s = (case generator g s of Done ⇒ None
    | Skip s' ⇒ force s'
    | Yield x s' ⇒ Some (x, s'))
by pat-completeness auto
termination
proof −
  have terminates (generator g) using generator[of g] by simp
  thus ?thesis by(rule terminates-wfE)(rule termination)
qed

lemma force-simps [simp]:
  generator g s = Done ⟹ force s = None
  generator g s = Skip s' ⟹ force s = force s'
  generator g s = Yield x s' ⟹ force s = Some (x, s')
by(simp-all)

declare force.simps[simp del]

lemma unstream-force-None [simp]: force s = None ⟹ unstream s = []
proof(induction s rule: force.induct)
  case (1 s)
  thus ?case by(cases generator g s) simp-all
qed

lemma unstream-force-Some [simp]: force s = Some (x, s') ⟹ unstream s = x #
unstream s'
```

proof(*induction s rule*: *force.induct*)
  case (*1 s*)
  thus *?case* **by**(*cases generator g s*) *simp-all*
qed

end

setup ‹*Context.theory-map* (*Stream-Fusion.add-unstream* @{*const-name unstream*})›

2.3 Producers

2.3.1 Conversion to streams

fun *stream-raw* :: *'a list* ⇒ (*'a*, *'a list*) *step*
where
  stream-raw [] = *Done*
| *stream-raw* (*x* # *xs*) = *Yield x xs*

lemma *terminates-stream-raw*: *terminates stream-raw*
proof (*rule terminatesI*)
  fix *s* :: *'a list*
  show *s* ∈ *terminates-on stream-raw*
    by(*induction s*)(*auto intro*: *terminates-on.intros*)
qed

lift-definition *stream* :: (*'a*, *'a list*) *generator* **is** *stream-raw* **by**(*rule terminates-stream-raw*)

lemma *unstream-stream*: *unstream stream xs* = *xs*
by(*induction xs*)(*auto simp add*: *stream.rep-eq*)

2.3.2 *replicate*

fun *replicate-raw* :: *'a* ⇒ (*'a*, *nat*) *raw-generator*
where
  replicate-raw a 0 = *Done*
| *replicate-raw a* (*Suc n*) = *Yield a n*

lemma *terminates-replicate-raw*: *terminates* (*replicate-raw a*)
proof (*rule terminatesI*)
  fix *s* :: *nat*
  show *s* ∈ *terminates-on* (*replicate-raw a*)
    by(*induction s*)(*auto intro*: *terminates-on.intros*)
qed

lift-definition *replicate-prod* :: *'a* ⇒ (*'a*, *nat*) *generator* **is** *replicate-raw*

by(*rule terminates-replicate-raw*)

lemma *unstream-replicate-prod* [*stream-fusion*]: *unstream* (*replicate-prod x*) *n* = *replicate n x*
by(*induction n*)(*simp-all add*: *replicate-prod.rep-eq*)

2.3.3 *upt*

definition *upt-raw* :: *nat* ⇒ (*nat*, *nat*) *raw-generator*
where *upt-raw n m* = (*if m* ≥ *n then Done else Yield m* (*Suc m*))

lemma *terminates-upt-raw*: *terminates* (*upt-raw n*)
proof (*rule terminatesI*)
fix *s* :: *nat*
show *s* ∈ *terminates-on* (*upt-raw n*)
by(*induction n*−*s arbitrary*: *s rule*: *nat.induct*)(*auto 4 3 simp add*: *upt-raw-def intro*: *terminates-on.intros*)
qed

lift-definition *upt-prod* :: *nat* ⇒ (*nat*, *nat*) *generator* **is** *upt-raw* **by**(*rule terminates-upt-raw*)

lemma *unstream-upt-prod* [*stream-fusion*]: *unstream* (*upt-prod n*) *m* = *upt m n*
by(*induction n*−*m arbitrary*: *n m*)(*simp-all add*: *upt-prod.rep-eq upt-conv-Cons upt-raw-def unstream.simps*)

2.3.4 *upto*

definition *upto-raw* :: *int* ⇒ (*int*, *int*) *raw-generator*
where *upto-raw n m* = (*if m* ≤ *n then Yield m* (*m* + *1*) *else Done*)

lemma *terminates-upto-raw*: *terminates* (*upto-raw n*)
proof (*rule terminatesI*)
fix *s* :: *int*
show *s* ∈ *terminates-on* (*upto-raw n*)
by(*induction nat*(*n*−*s*+*1*) *arbitrary*: *s*)(*auto 4 3 simp add*: *upto-raw-def intro*: *terminates-on.intros*)
qed

lift-definition *upto-prod* :: *int* ⇒ (*int*, *int*) *generator* **is** *upto-raw* **by** (*rule terminates-upto-raw*)

lemma *unstream-upto-prod* [*stream-fusion*]: *unstream* (*upto-prod n*) *m* = *upto m n*
by(*induction nat* (*n* − *m* + *1*) *arbitrary*: *m*)(*simp-all add*: *upto-prod.rep-eq upto.simps upto-raw-def*)

2.3.5 []

lift-definition *Nil-prod* :: ('*a*, *unit*) *generator* **is** λ-. *Done*
by(*auto simp add*: *terminates-def intro*: *terminates-on.intros*)

lemma *generator-Nil-prod*: *generator Nil-prod* = (λ-. *Done*)
by(*fact Nil-prod.rep-eq*)

lemma *unstream-Nil-prod* [*stream-fusion*]: *unstream Nil-prod* () = []
by(*simp add*: *generator-Nil-prod*)

2.4 Consumers

2.4.1 (!)

context fixes *g* :: ('*a*, '*s*) *generator* **begin**

definition *nth-cons* :: '*s* ⇒ *nat* ⇒ '*a*
where [*stream-fusion*]: *nth-cons s n* = *unstream g s* ! *n*

lemma *nth-cons-code* [*code*]:
nth-cons s n =
(*case generator g s of Done* => *undefined n*
| *Skip s'* => *nth-cons s' n*
| *Yield x s'* => (*case n of 0* => *x* | *Suc n'* => *nth-cons s' n'*))
by(*cases generator g s*)(*simp-all add*: *nth-cons-def nth-def split*: *nat.split*)

end

2.4.2 *length*

context fixes *g* :: ('*a*, '*s*) *generator* **begin**

definition *length-cons* :: '*s* ⇒ *nat*
where *length-cons s* = *length* (*unstream g s*)

lemma *length-cons-code* [*code*]:
length-cons s =
(*case generator g s of*
Done ⇒ *0*
| *Skip s'* ⇒ *length-cons s'*
| *Yield a s'* ⇒ *1* + *length-cons s'*)
by(*cases generator g s*)(*simp-all add*: *length-cons-def*)

definition *gen-length-cons* :: *nat* ⇒ '*s* ⇒ *nat*

where *gen-length-cons n s = n + length (unstream g s)*

lemma *gen-length-cons-code* [*code*]:
gen-length-cons n s = (case generator g s of
Done ⇒ n | Skip s′ ⇒ gen-length-cons n s′ | Yield a s′ ⇒ gen-length-cons (Suc n) s′)
by(*simp add*: *gen-length-cons-def split*: *step.split*)

lemma *unstream-gen-length* [*stream-fusion*]: *gen-length-cons 0 s = length (unstream g s)*
by(*simp add*: *gen-length-cons-def*)

lemma *unstream-gen-length2* [*stream-fusion*]: *gen-length-cons n s = List.gen-length n (unstream g s)*
by(*simp add*: *List.gen-length-def gen-length-cons-def*)

end

2.4.3 *foldr*

context
fixes *g* :: (*′a*, *′s*) *generator*
and *f* :: *′a ⇒ ′b ⇒ ′b*
and *z* :: *′b*
begin

definition *foldr-cons* :: *′s ⇒ ′b*
where [*stream-fusion*]: *foldr-cons s = foldr f (unstream g s) z*

lemma *foldr-cons-code* [*code*]:
foldr-cons s =
(case generator g s of
Done ⇒ z
| Skip s′ ⇒ foldr-cons s′
| Yield a s′ ⇒ f a (foldr-cons s′))
by(*cases generator g s*)(*simp-all add*: *foldr-cons-def*)

end

2.4.4 *foldl*

context
fixes *g* :: (*′b*, *′s*) *generator*
and *f* :: *′a ⇒ ′b ⇒ ′a*

begin

definition *foldl-cons* :: *'a* ⇒ *'s* ⇒ *'a*
where [*stream-fusion*]: *foldl-cons z s* = *foldl f z* (*unstream g s*)

lemma *foldl-cons-code* [*code*]:
foldl-cons z s =
(*case generator g s of*
Done ⇒ *z*
| *Skip s'* ⇒ *foldl-cons z s'*
| *Yield a s'* ⇒ *foldl-cons* (*f z a*) *s'*)
by (*cases generator g s*)(*simp-all add*: *foldl-cons-def*)

end

2.4.5 *fold*

context
fixes *g* :: (*'a*, *'s*) *generator*
and *f* :: *'a* ⇒ *'b* ⇒ *'b*
begin

definition *fold-cons* :: *'b* ⇒ *'s* ⇒ *'b*
where [*stream-fusion*]: *fold-cons z s* = *fold f* (*unstream g s*) *z*

lemma *fold-cons-code* [*code*]:
fold-cons z s =
(*case generator g s of*
Done ⇒ *z*
| *Skip s'* ⇒ *fold-cons z s'*
| *Yield a s'* ⇒ *fold-cons* (*f a z*) *s'*)
by (*cases generator g s*)(*simp-all add*: *fold-cons-def*)

end

2.4.6 *List.null*

definition *null-cons* :: (*'a*, *'s*) *generator* ⇒ *'s* ⇒ *bool*
where [*stream-fusion*]: *null-cons g s* = *List.null* (*unstream g s*)

lemma *null-cons-code* [*code*]:
null-cons g s = (*case generator g s of Done* ⇒ *True* | *Skip s'* ⇒ *null-cons g s'* | *Yield*
- - ⇒ *False*)
by(*cases generator g s*)(*simp-all add*: *null-cons-def null-def*)

2.4.7 *hd*

context fixes $g :: ('a, 's)$ *generator* **begin**

definition *hd-cons* :: $'s \Rightarrow 'a$
where [*stream-fusion*]: *hd-cons* s = *hd* (*unstream* g s)

lemma *hd-cons-code* [*code*]:
hd-cons s =
(*case generator* g s *of*
Done ⇒ *undefined*
| *Skip* s' ⇒ *hd-cons* s'
| *Yield* a s' ⇒ a)
by (*cases generator* g s)(*simp-all add*: *hd-cons-def hd-def*)

end

2.4.8 *last*

context fixes $g :: ('a, 's)$ *generator* **begin**

definition *last-cons* :: $'a$ *option* ⇒ $'s \Rightarrow 'a$
where *last-cons* x s = (*if unstream* g s = [] *then the* x *else last* (*unstream* g s))

lemma *last-cons-code* [*code*]:
last-cons x s =
(*case generator* g s *of Done* ⇒ *the* x
| *Skip* s' ⇒ *last-cons* x s'
| *Yield* a s' ⇒ *last-cons* (*Some* a) s')
by (*cases generator* g s)(*simp-all add*: *last-cons-def*)

lemma *unstream-last-cons* [*stream-fusion*]: *last-cons None* s = *last* (*unstream* g s)
by (*simp add*: *last-cons-def last-def option.the-def*)

end

2.4.9 *sum-list*

context fixes $g :: ('a :: monoid\text{-}add, 's)$ *generator* **begin**

definition *sum-list-cons* :: $'s \Rightarrow 'a$
where [*stream-fusion*]: *sum-list-cons* s = *sum-list* (*unstream* g s)

lemma *sum-list-cons-code* [*code*]:
sum-list-cons s =

(*case generator g s of*
Done ⇒ *0*
| *Skip s′* ⇒ *sum-list-cons s′*
| *Yield a s′* ⇒ *a* + *sum-list-cons s′*)
by (*cases generator g s*)(*simp-all add*: *sum-list-cons-def*)

end

2.4.10 *list-all2*

context
fixes *g* :: (′*a*, ′*s1*) *generator*
and *h* :: (′*b*, ′*s2*) *generator*
and *P* :: ′*a* ⇒ ′*b* ⇒ *bool*
begin

definition *list-all2-cons* :: ′*s1* ⇒ ′*s2* ⇒ *bool*
where [*stream-fusion*]: *list-all2-cons sg sh* = *list-all2 P* (*unstream g sg*) (*unstream h sh*)

definition *list-all2-cons1* :: ′*a* ⇒ ′*s1* ⇒ ′*s2* ⇒ *bool*
where *list-all2-cons1 x sg′ sh* = *list-all2 P* (*x* # *unstream g sg′*) (*unstream h sh*)

lemma *list-all2-cons-code* [*code*]:
list-all2-cons sg sh =
(*case generator g sg of*
Done ⇒ *null-cons h sh*
| *Skip sg′* ⇒ *list-all2-cons sg′ sh*
| *Yield a sg′* ⇒ *list-all2-cons1 a sg′ sh*)
by(*simp split*: *step.split add*: *list-all2-cons-def null-cons-def List.null-def list-all2-cons1-def*)

lemma *list-all2-cons1-code* [*code*]:
list-all2-cons1 x sg′ sh =
(*case generator h sh of*
Done ⇒ *False*
| *Skip sh′* ⇒ *list-all2-cons1 x sg′ sh′*
| *Yield y sh′* ⇒ *P x y* ∧ *list-all2-cons sg′ sh′*)
by(*simp split*: *step.split add*: *list-all2-cons-def null-cons-def List.null-def list-all2-cons1-def*)

end

2.4.11 *list-all*

context
fixes *g* :: (′*a*, ′*s*) *generator*

and $P :: 'a \Rightarrow bool$
begin

definition *list-all-cons* :: $'s \Rightarrow bool$
where [*stream-fusion*]: *list-all-cons s* = *list-all P* (*unstream g s*)

lemma *list-all-cons-code* [*code*]:
list-all-cons s $\longleftrightarrow$
(*case generator g s of*
Done $\Rightarrow$ *True* | *Skip s'* $\Rightarrow$ *list-all-cons s'* | *Yield x s'* $\Rightarrow$ *P x* $\wedge$ *list-all-cons s'*)
by(*simp add*: *list-all-cons-def split*: *step.split*)

end

2.4.12 *ord.lexordp*

context *ord* **begin**

definition *lexord-fusion* :: ('*a*, '*s1*) *generator* $\Rightarrow$ ('*a*, '*s2*) *generator* $\Rightarrow$ '*s1* $\Rightarrow$ '*s2* $\Rightarrow$ *bool*
where [*code del*]: *lexord-fusion g1 g2 s1 s2* = *ord-class.lexordp* (*unstream g1 s1*) (*unstream g2 s2*)

definition *lexord-eq-fusion* :: ('*a*, '*s1*) *generator* $\Rightarrow$ ('*a*, '*s2*) *generator* $\Rightarrow$ '*s1* $\Rightarrow$ '*s2* $\Rightarrow$ *bool*
where [*code del*]: *lexord-eq-fusion g1 g2 s1 s2* = *lexordp-eq* (*unstream g1 s1*) (*unstream g2 s2*)

lemma *lexord-fusion-code*:
lexord-fusion g1 g2 s1 s2 $\longleftrightarrow$
(*case generator g1 s1 of*
Done $\Rightarrow$ $\neg$ *null-cons g2 s2*
| *Skip s1'* $\Rightarrow$ *lexord-fusion g1 g2 s1' s2*
| *Yield x s1'* $\Rightarrow$
(*case force g2 s2 of*
None $\Rightarrow$ *False*
| *Some* (*y*, *s2'*) $\Rightarrow$ *x* < *y* $\vee$ $\neg$ *y* < *x* $\wedge$ *lexord-fusion g1 g2 s1' s2'*))
unfolding *lexord-fusion-def*
by(*cases generator g1 s1 force g2 s2 rule*: *step.exhaust*[*case-product option.exhaust*])(*auto simp add*: *null-cons-def null-def*)

lemma *lexord-eq-fusion-code*:
lexord-eq-fusion g1 g2 s1 s2 $\longleftrightarrow$
(*case generator g1 s1 of*

```
    Done ⇒ True
  | Skip s1' ⇒ lexord-eq-fusion g1 g2 s1' s2
  | Yield x s1' ⇒
    (case force g2 s2 of
       None ⇒ False
     | Some (y, s2') ⇒ x < y ∨ ¬ y < x ∧ lexord-eq-fusion g1 g2 s1' s2'))
unfolding lexord-eq-fusion-def
by(cases generator g1 s1 force g2 s2 rule: step.exhaust[case-product option.exhaust]) auto

end

lemmas [code] =
  lexord-fusion-code ord.lexord-fusion-code
  lexord-eq-fusion-code ord.lexord-eq-fusion-code

lemmas [stream-fusion] =
  lexord-fusion-def ord.lexord-fusion-def
  lexord-eq-fusion-def ord.lexord-eq-fusion-def
```

2.5 Transformers

2.5.1 *map*

```
definition map-raw :: ('a ⇒ 'b) ⇒ ('a, 's) raw-generator ⇒ ('b, 's) raw-generator
where
  map-raw f g s = (case g s of
     Done ⇒ Done
   | Skip s' ⇒ Skip s'
   | Yield a s' ⇒ Yield (f a) s')

lemma terminates-map-raw:
  assumes terminates g
  shows terminates (map-raw f g)
proof (rule terminatesI)
  fix s
  from assms
  have s ∈ terminates-on g by (simp add: terminates-def)
  then show s ∈ terminates-on (map-raw f g)
    by (induction s)(auto intro: terminates-on.intros simp add: map-raw-def)
qed

lift-definition map-trans :: ('a ⇒ 'b) ⇒ ('a, 's) generator ⇒ ('b, 's) generator is
map-raw
by (rule terminates-map-raw)
```

lemma *unstream-map-trans* [*stream-fusion*]: *unstream* (*map-trans f g*) *s* = *map f* (*unstream g s*)
proof (*induction s taking*: *g rule*: *unstream.induct*)
case (*1 s*)
show *?case* **using** *1.IH* **by** (*cases generator g s*)(*simp-all add*: *map-trans.rep-eq map-raw-def*)
qed

2.5.2 *drop*

fun *drop-raw* :: ('*a*, '*s*) *raw-generator* ⇒ ('*a*, (*nat* × '*s*)) *raw-generator*
where
drop-raw g (*n*, *s*) = (*case g s of*
Done ⇒ *Done* | *Skip s'* ⇒ *Skip* (*n*, *s'*)
| *Yield a s'* ⇒ (*case n of 0* ⇒ *Yield a* (*0*, *s'*) | *Suc n* ⇒ *Skip* (*n*, *s'*)))

lemma *terminates-drop-raw*:
assumes *terminates g*
shows *terminates* (*drop-raw g*)
proof (*rule terminatesI*)
fix *st* :: *nat* × '*a*
obtain *n s* **where** *st* = (*n*, *s*) **by**(*cases st*)
from *assms* **have** *s* ∈ *terminates-on g* **by** (*simp add*: *terminates-def*)
thus *st* ∈ *terminates-on* (*drop-raw g*) **unfolding** ‹*st* = (*n*, *s*)›
apply(*induction arbitrary*: *n*)
apply(*case-tac* [!] *n*)
apply(*auto intro*: *terminates-on.intros*)
done
qed

lift-definition *drop-trans* :: ('*a*, '*s*) *generator* ⇒ ('*a*, *nat* × '*s*) *generator* **is** *drop-raw*
by (*rule terminates-drop-raw*)

lemma *unstream-drop-trans* [*stream-fusion*]: *unstream* (*drop-trans g*) (*n*, *s*) = *drop n* (*unstream g s*)
proof (*induction s arbitrary*: *n taking*: *g rule*: *unstream.induct*)
case (*1 s*)
show *?case* **using** *1.IH*(*1*)[*of - n*] *1.IH*(*2*)[*of - - n*] *1.IH*(*2*)[*of - - n − 1*]
by(*cases generator g s n rule*: *step.exhaust*[*case-product nat.exhaust*])
(*simp-all add*: *drop-trans.rep-eq*)
qed

2.5.3 *dropWhile*

fun *dropWhile-raw* :: $('a \Rightarrow bool) \Rightarrow ('a, 's)$ *raw-generator* $\Rightarrow ('a, bool \times 's)$ *raw-generator*
— Boolean flag indicates whether we are still in dropping phase
where
dropWhile-raw P g (*True*, *s*) = (*case g s of*
Done ⇒ *Done* | *Skip s′* ⇒ *Skip* (*True*, *s′*)
| *Yield a s′* ⇒ (*if P a then Skip* (*True*, *s′*) *else Yield a* (*False*, *s′*)))
| *dropWhile-raw P g* (*False*, *s*) = (*case g s of*
Done ⇒ *Done* | *Skip s′* ⇒ *Skip* (*False*, *s′*) | *Yield a s′* ⇒ *Yield a* (*False*, *s′*))

lemma *terminates-dropWhile-raw*:
assumes *terminates g*
shows *terminates* (*dropWhile-raw P g*)
proof (*rule terminatesI*)
fix *st* :: *bool* × *′a*
obtain *b s* **where** *st* = (*b*, *s*) **by** (*cases st*)
from *assms* **have** *s* ∈ *terminates-on g* **by** (*simp add*: *terminates-def*)
then show *st* ∈ *terminates-on* (*dropWhile-raw P g*) **unfolding** ‹*st* = (*b*, *s*)›
proof (*induction s arbitrary*: *b*)
case (*stop s b*)
then show *?case* **by** (*cases b*)(*simp-all add*: *terminates-on.stop*)
next
case (*pause s s′ b*)
then show *?case* **by** (*cases b*)(*simp-all add*: *terminates-on.pause*)
next
case (*unfold s a s′ b*)
then show *?case*
by(*cases b*)(*cases P a*, *auto intro*: *terminates-on.pause terminates-on.unfold*)
qed
qed

lift-definition *dropWhile-trans* :: $('a \Rightarrow bool) \Rightarrow ('a, 's)$ *generator* $\Rightarrow ('a, bool \times 's)$ *generator*
is *dropWhile-raw* **by** (*rule terminates-dropWhile-raw*)

lemma *unstream-dropWhile-trans-False*:
unstream (*dropWhile-trans P g*) (*False*, *s*) = *unstream g s*
proof (*induction s taking*: *g rule*: *unstream.induct*)
case (*1 s*)
then show *?case* **by** (*cases generator g s*)(*simp-all add*: *dropWhile-trans.rep-eq*)
qed

lemma *unstream-dropWhile-trans* [*stream-fusion*]:

unstream (*dropWhile-trans P g*) (*True*, *s*) = *dropWhile P* (*unstream g s*)
proof (*induction s taking*: *g rule*: *unstream.induct*)
case (*1 s*)
then show *?case*
proof(*cases generator g s*)
case (*Yield a s'*)
then show *?thesis* **using** *1.IH*(*2*) *unstream-dropWhile-trans-False*
by (*cases P a*)(*simp-all add*: *dropWhile-trans.rep-eq*)
qed(*simp-all add*: *dropWhile-trans.rep-eq*)
qed

2.5.4 *take*

fun *take-raw* :: ('*a*, '*s*) *raw-generator* ⇒ ('*a*, (*nat* × '*s*)) *raw-generator*
where
take-raw g (*0*, *s*) = *Done*
| *take-raw g* (*Suc n*, *s*) = (*case g s of*
Done ⇒ *Done* | *Skip s'* ⇒ *Skip* (*Suc n*, *s'*) | *Yield a s'* ⇒ *Yield a* (*n*, *s'*))

lemma *terminates-take-raw*:
assumes *terminates g*
shows *terminates* (*take-raw g*)
proof (*rule terminatesI*)
fix *st* :: *nat* × '*a*
obtain *n s* **where** *st* = (*n*, *s*) **by**(*cases st*)
from *assms* **have** *s* ∈ *terminates-on g* **by** (*simp add*: *terminates-def*)
thus *st* ∈ *terminates-on* (*take-raw g*) **unfolding** ‹*st* = (*n*, *s*)›
apply(*induction s arbitrary*: *n*)
apply(*case-tac* [!] *n*)
apply(*auto intro*: *terminates-on.intros*)
done
qed

lift-definition *take-trans* :: ('*a*, '*s*) *generator* ⇒ ('*a*, *nat* × '*s*) *generator* **is** *take-raw*
by (*rule terminates-take-raw*)

lemma *unstream-take-trans* [*stream-fusion*]: *unstream* (*take-trans g*) (*n*, *s*) = *take n* (*unstream g s*)
proof (*induction s arbitrary*: *n taking*: *g rule*: *unstream.induct*)
case (*1 s*)
show *?case* **using** *1.IH*(*1*)[*of - n*] *1.IH*(*2*)
by(*cases generator g s n rule*: *step.exhaust*[*case-product nat.exhaust*])
(*simp-all add*: *take-trans.rep-eq*)
qed

2.5.5 *takeWhile*

definition *takeWhile-raw* :: ('*a* ⇒ *bool*) ⇒ ('*a*, '*s*) *raw-generator* ⇒ ('*a*, '*s*) *raw-generator*
where
takeWhile-raw P g s = (*case g s of*
Done ⇒ *Done* | *Skip s'* ⇒ *Skip s'* | *Yield a s'* ⇒ *if P a then Yield a s' else Done*)

lemma *terminates-takeWhile-raw*:
assumes *terminates g*
shows *terminates* (*takeWhile-raw P g*)
proof (*rule terminatesI*)
fix *s*
from *assms* **have** *s* ∈ *terminates-on g* **by** (*simp add*: *terminates-def*)
thus *s* ∈ *terminates-on* (*takeWhile-raw P g*)
proof (*induction s rule*: *terminates-on.induct*)
case (*unfold s a s'*)
then show *?case* **by**(*cases P a*)(*auto simp add*: *takeWhile-raw-def intro*: *terminates-on.intros*)
qed(*auto intro*: *terminates-on.intros simp add*: *takeWhile-raw-def*)
qed

lift-definition *takeWhile-trans* :: ('*a* ⇒ *bool*) ⇒ ('*a*, '*s*) *generator* ⇒ ('*a*, '*s*) *generator*
is *takeWhile-raw* **by** (*rule terminates-takeWhile-raw*)

lemma *unstream-takeWhile-trans* [*stream-fusion*]:
unstream (*takeWhile-trans P g*) *s* = *takeWhile P* (*unstream g s*)
proof (*induction s taking*: *g rule*: *unstream.induct*)
case (*1 s*)
then show *?case* **by**(*cases generator g s*)(*simp-all add*: *takeWhile-trans.rep-eq takeWhile-raw-def*)
qed

2.5.6 (@)

fun *append-raw* :: ('*a*, '*sg*) *raw-generator* ⇒ ('*a*, '*sh*) *raw-generator* ⇒ '*sh* ⇒ ('*a*, '*sg* + '*sh*) *raw-generator*
where
append-raw g h sh-start (*Inl sg*) = (*case g sg of*
Done ⇒ *Skip* (*Inr sh-start*) | *Skip sg'* ⇒ *Skip* (*Inl sg'*) | *Yield a sg'* ⇒ *Yield a* (*Inl sg'*))
| *append-raw g h sh-start* (*Inr sh*) = (*case h sh of*
Done ⇒ *Done* | *Skip sh'* ⇒ *Skip* (*Inr sh'*) | *Yield a sh'* ⇒ *Yield a* (*Inr sh'*))

lemma *terminates-on-append-raw-Inr*:

assumes *terminates h*
shows *Inr sh* ∈ *terminates-on* (*append-raw g h sh-start*)
proof −
from *assms* **have** *sh* ∈ *terminates-on h* **by** (*simp add*: *terminates-def*)
thus *?thesis* **by**(*induction sh*)(*auto intro*: *terminates-on.intros*)
qed

lemma *terminates-append-raw*:
assumes *terminates g terminates h*
shows *terminates* (*append-raw g h sh-start*)
proof (*rule terminatesI*)
fix *s*
show *s* ∈ *terminates-on* (*append-raw g h sh-start*)
proof (*cases s*)
case (*Inl sg*)
from ‹*terminates g*› **have** *sg* ∈ *terminates-on g* **by** (*simp add*: *terminates-def*)
thus *s* ∈ *terminates-on* (*append-raw g h sh-start*) **unfolding** *Inl*
by *induction*(*auto intro*: *terminates-on.intros terminates-on-append-raw-Inr*[*OF* ‹*terminates h*›])
qed(*simp add*: *terminates-on-append-raw-Inr*[*OF* ‹*terminates h*›])
qed

lift-definition *append-trans* :: ('*a*, '*sg*) *generator* ⇒ ('*a*, '*sh*) *generator* ⇒ '*sh* ⇒ ('*a*, '*sg* + '*sh*) *generator*
is *append-raw* **by** (*rule terminates-append-raw*)

lemma *unstream-append-trans-Inr*: *unstream* (*append-trans g h sh*) (*Inr sh*′) = *unstream h sh*′
proof (*induction sh*′ *taking*: *h rule*: *unstream.induct*)
case (*1 sh*′)
then show *?case* **by** (*cases generator h sh*′)(*simp-all add*: *append-trans.rep-eq*)
qed

lemma *unstream-append-trans* [*stream-fusion*]:
unstream (*append-trans g h sh*) (*Inl sg*) = *append* (*unstream g sg*) (*unstream h sh*)
proof(*induction sg taking*: *g rule*: *unstream.induct*)
case (*1 sg*)
then show *?case* **using** *unstream-append-trans-Inr*
by (*cases generator g sg*)(*simp-all add*: *append-trans.rep-eq*)
qed

2.5.7 *filter*

definition *filter-raw* :: ('*a* ⇒ *bool*) ⇒ ('*a*, '*s*) *raw-generator* ⇒ ('*a*, '*s*) *raw-generator*

where
filter-raw P g s = (*case g s of*
Done ⇒ *Done* | *Skip s′* ⇒ *Skip s′* | *Yield a s′* ⇒ *if P a then Yield a s′ else Skip s′*)

lemma *terminates-filter-raw*:
assumes *terminates g*
shows *terminates* (*filter-raw P g*)
proof (*rule terminatesI*)
fix *s*
from *assms* **have** *s* ∈ *terminates-on g* **by** (*simp add*: *terminates-def*)
thus *s* ∈ *terminates-on* (*filter-raw P g*)
proof(*induction s*)
case (*unfold s a s′*)
thus *?case*
by(*cases P a*)(*auto intro*: *terminates-on.intros simp add*: *filter-raw-def*)
qed(*auto intro*: *terminates-on.intros simp add*: *filter-raw-def*)
qed

lift-definition *filter-trans* :: (*′a* ⇒ *bool*) ⇒ (*′a*,*′s*) *generator* ⇒ (*′a*,*′s*) *generator*
is *filter-raw* **by** (*rule terminates-filter-raw*)

lemma *unstream-filter-trans* [*stream-fusion*]: *unstream* (*filter-trans P g*) *s* = *filter P* (*unstream g s*)
proof (*induction s taking*: *g rule*: *unstream.induct*)
case (*1 s*)
then show *?case* **by**(*cases generator g s*)(*simp-all add*: *filter-trans.rep-eq filter-raw-def*)
qed

2.5.8 *zip*

fun *zip-raw* :: (*′a*, *′sg*) *raw-generator* ⇒ (*′b*, *′sh*) *raw-generator* ⇒ (*′a* × *′b*, *′sg* × *′sh* × *′a option*) *raw-generator*
— We search first the left list for the next element and cache it in the *′a option* part of the state once we found one
where
zip-raw g h (*sg*, *sh*, *None*) = (*case g sg of*
Done ⇒ *Done* | *Skip sg′* ⇒ *Skip* (*sg′*, *sh*, *None*) | *Yield a sg′* ⇒ *Skip* (*sg′*, *sh*, *Some a*))
| *zip-raw g h* (*sg*, *sh*, *Some a*) = (*case h sh of*
Done ⇒ *Done* | *Skip sh′* ⇒ *Skip* (*sg*, *sh′*, *Some a*) | *Yield b sh′* ⇒ *Yield* (*a*, *b*) (*sg*, *sh′*, *None*))

lemma *terminates-zip-raw*:
assumes *terminates g terminates h*

```
  shows terminates (zip-raw g h)
proof (rule terminatesI)
  fix s :: 'a × 'c × 'b option
  obtain sg sh m where s = (sg, sh, m) by(cases s)
  show s ∈ terminates-on (zip-raw g h)
  proof(cases m)
    case None
    from ‹terminates g› have sg ∈ terminates-on g by (simp add: terminates-def)
    then show ?thesis unfolding ‹s = (sg, sh, m)› None
    proof (induction sg arbitrary: sh)
      case (unfold sg a sg')
      from ‹terminates h› have sh ∈ terminates-on h by (simp add: terminates-def)
      hence (sg', sh, Some a) ∈ terminates-on (zip-raw g h)
        by induction(auto intro: terminates-on.intros unfold.IH)
      thus ?case using unfold.hyps by(auto intro: terminates-on.pause)
    qed(simp-all add: terminates-on.stop terminates-on.pause)
  next
    case (Some a')
    from ‹terminates h› have sh ∈ terminates-on h by (simp add: terminates-def)
    thus ?thesis unfolding ‹s = (sg, sh, m)› Some
    proof (induction sh arbitrary: sg a')
      case (unfold sh b sh')
      from ‹terminates g› have sg ∈ terminates-on g by (simp add: terminates-def)
      hence (sg, sh', None) ∈ terminates-on (zip-raw g h)
        by induction(auto intro: terminates-on.intros unfold.IH)
      thus ?case using unfold.hyps by(auto intro: terminates-on.unfold)
    qed(simp-all add: terminates-on.stop terminates-on.pause)
  qed
qed

lift-definition zip-trans :: ('a, 'sg) generator ⇒ ('b, 'sh) generator ⇒ ('a × 'b,'sg ×
'sh × 'a option) generator
is zip-raw by (rule terminates-zip-raw)

lemma unstream-zip-trans [stream-fusion]:
  unstream (zip-trans g h) (sg, sh, None) = zip (unstream g sg) (unstream h sh)
proof (induction sg arbitrary: sh taking: g rule: unstream.induct)
  case (1 sg)
  then show ?case
  proof (cases generator g sg)
    case (Yield a sg')
    note IH = 1.IH(2)[OF Yield]
     have unstream (zip-trans g h) (sg', sh, Some a) = zip (a # (unstream g sg'))
(unstream h sh)
```

proof(*induction sh taking*: *h rule*: *unstream.induct*)
case (*1 sh*)
then show *?case* **using** *IH* **by**(*cases generator h sh*)(*simp-all add*: *zip-trans.rep-eq*)
qed
then show *?thesis* **using** *Yield* **by** (*simp add*: *zip-trans.rep-eq*)
qed(*simp-all add*: *zip-trans.rep-eq*)
qed

2.5.9 *tl*

fun *tl-raw* :: (*'a*, *'sg*) *raw-generator* ⇒ (*'a*, *bool* × *'sg*) *raw-generator*
— The Boolean flag stores whether we have already skipped the first element
where
tl-raw g (*False*, *sg*) = (*case g sg of*
Done ⇒ *Done* | *Skip sg'* ⇒ *Skip* (*False*, *sg'*) | *Yield a sg'* ⇒ *Skip* (*True*,*sg'*))
| *tl-raw g* (*True*, *sg*) = (*case g sg of*
Done ⇒ *Done* | *Skip sg'* ⇒ *Skip* (*True*, *sg'*) | *Yield a sg'* ⇒ *Yield a* (*True*, *sg'*))

lemma *terminates-tl-raw*:
assumes *terminates g*
shows *terminates* (*tl-raw g*)
proof (*rule terminatesI*)
fix *s* :: *bool* × *'a*
obtain *b sg* **where** *s* = (*b*, *sg*) **by**(*cases s*)
{ fix *sg*
from *assms* **have** *sg* ∈ *terminates-on g* **by**(*simp add*: *terminates-def*)
hence (*True*, *sg*) ∈ *terminates-on* (*tl-raw g*)
by(*induction sg*)(*auto intro*: *terminates-on.intros*) **}**
moreover from *assms* **have** *sg* ∈ *terminates-on g* **by**(*simp add*: *terminates-def*)
hence (*False*, *sg*) ∈ *terminates-on* (*tl-raw g*)
by(*induction sg*)(*auto intro*: *terminates-on.intros calculation*)
ultimately show *s* ∈ *terminates-on* (*tl-raw g*) **using** ‹*s* = (*b*, *sg*)›
by(*cases b*) *simp-all*
qed

lift-definition *tl-trans* :: (*'a*, *'sg*) *generator* ⇒ (*'a*, *bool* × *'sg*) *generator*
is *tl-raw* **by**(*rule terminates-tl-raw*)

lemma *unstream-tl-trans-True*: *unstream* (*tl-trans g*) (*True*, *s*) = *unstream g s*
proof(*induction s taking*: *g rule*: *unstream.induct*)
case (*1 s*)
show *?case* **using** *1.IH* **by** (*cases generator g s*)(*simp-all add*: *tl-trans.rep-eq*)
qed

lemma *unstream-tl-trans* [*stream-fusion*]: *unstream* (*tl-trans g*) (*False*, *s*) = *tl* (*unstream g s*)
proof (*induction s taking*: *g rule*: *unstream.induct*)
  case (*1 s*)
  then show *?case* **using** *unstream-tl-trans-True*
    by (*cases generator g s*)(*simp-all add*: *tl-trans.rep-eq*)
qed

2.5.10 *butlast*

fun *butlast-raw* :: (*'a*, *'s*) *raw-generator* ⇒ (*'a*, *'a option* × *'s*) *raw-generator*
  — The *'a option* caches the previous element we have seen
where
  butlast-raw g (*None*,*s*) = (*case g s of*
    Done ⇒ *Done* | *Skip s'* ⇒ *Skip* (*None*, *s'*) | *Yield a s'* ⇒ *Skip* (*Some a*, *s'*))
| *butlast-raw g* (*Some b*, *s*) = (*case g s of*
    Done ⇒ *Done* | *Skip s'* ⇒ *Skip* (*Some b*, *s'*) | *Yield a s'* ⇒ *Yield b* (*Some a*, *s'*))

lemma *terminates-butlast-raw*:
  assumes *terminates g*
  shows *terminates* (*butlast-raw g*)
proof (*rule terminatesI*)
  fix *st* :: *'b option* × *'a*
  obtain *ma s* **where** *st* = (*ma*,*s*) **by** (*cases st*)
  from *assms* **have** *s* ∈ *terminates-on g* **by** (*simp add*: *terminates-def*)
  then show *st* ∈ *terminates-on* (*butlast-raw g*) **unfolding** ‹*st* = (*ma*, *s*)›
    apply(*induction s arbitrary*: *ma*)
    apply(*case-tac* [!] *ma*)
    apply(*auto intro*: *terminates-on.intros*)
    done
qed

lift-definition *butlast-trans* :: (*'a*,*'s*) *generator* ⇒ (*'a*, *'a option* × *'s*) *generator*
is *butlast-raw* **by** (*rule terminates-butlast-raw*)

lemma *unstream-butlast-trans-Some*:
  unstream (*butlast-trans g*) (*Some b*,*s*) = *butlast* (*b* # (*unstream g s*))
proof (*induction s arbitrary*: *b taking*: *g rule*: *unstream.induct*)
  case (*1 s*)
  then show *?case* **by** (*cases generator g s*)(*simp-all add*: *butlast-trans.rep-eq*)
qed

lemma *unstream-butlast-trans* [*stream-fusion*]:
  unstream (*butlast-trans g*) (*None*, *s*) = *butlast* (*unstream g s*)

proof (*induction s taking*: *g rule*: *unstream.induct*)
  case (*1 s*)
  then show *?case* **using** *1 unstream-butlast-trans-Some*[*of g*]
    by (*cases generator g s*)(*simp-all add*: *butlast-trans.rep-eq*)
qed

2.5.11 *concat*

We only do the easy version here where the generator has type (*'a list*, *'s*) *generator*, not ((*'a*, *'si*) *generator*, *'s*) *generator*

fun *concat-raw* :: (*'a list*, *'s*) *raw-generator* ⇒ (*'a*, *'a list* × *'s*) *raw-generator*
where
  concat-raw g ([], *s*) = (*case g s of*
    Done ⇒ *Done* | *Skip s'* ⇒ *Skip* ([], *s'*) | *Yield xs s'* ⇒ *Skip* (*xs*, *s'*))
| *concat-raw g* (*x* # *xs*, *s*) = *Yield x* (*xs*, *s*)

lemma *terminates-concat-raw*:
  assumes *terminates g*
  shows *terminates* (*concat-raw g*)
proof (*rule terminatesI*)
  fix *st* :: *'b list* × *'a*
  obtain *xs s* **where** *st* = (*xs*, *s*) **by** (*cases st*)
  from *assms* **have** *s* ∈ *terminates-on g* **by** (*simp add*: *terminates-def*)
  then show *st* ∈ *terminates-on* (*concat-raw g*) **unfolding** ‹*st* = (*xs*, *s*)›
  proof (*induction s arbitrary*: *xs*)
    case (*stop s xs*)
   then show *?case* **by** (*induction xs*)(*auto intro*: *terminates-on.stop terminates-on.unfold*)
  next
    case (*pause s s' xs*)
   then show *?case* **by** (*induction xs*)(*auto intro*: *terminates-on.pause terminates-on.unfold*)
  next
    case (*unfold s a s' xs*)
   then show *?case* **by** (*induction xs*)(*auto intro*: *terminates-on.pause terminates-on.unfold*)
  qed
qed

lift-definition *concat-trans* :: (*'a list*, *'s*) *generator* ⇒ (*'a*, *'a list* × *'s*) *generator*
is *concat-raw* **by** (*rule terminates-concat-raw*)

lemma *unstream-concat-trans-gen*: *unstream* (*concat-trans g*) (*xs*, *s*) = *xs* @ (*concat* (*unstream g s*))
proof (*induction s arbitrary*: *xs taking*: *g rule*: *unstream.induct*)
  case (*1 s*)

then show *unstream* (*concat-trans g*) (*xs*, *s*) = *xs* @ (*concat* (*unstream g s*))
proof (*cases generator g s*)
case *Done*
then show *?thesis* **by** (*induction xs*)(*simp-all add*: *concat-trans.rep-eq*)
next
case (*Skip s'*)
then show *?thesis* **using** *1.IH*(*1*)[*of s' Nil*]
by (*induction xs*)(*simp-all add*: *concat-trans.rep-eq*)
next
case (*Yield a s'*)
then show *?thesis* **using** *1.IH*(*2*)[*of a s' a*]
by (*induction xs*)(*simp-all add*: *concat-trans.rep-eq*)
qed
qed

lemma *unstream-concat-trans* [*stream-fusion*]:
unstream (*concat-trans g*) ([], *s*) = *concat* (*unstream g s*)
by(*simp only*: *unstream-concat-trans-gen append-Nil*)

2.5.12 *splice*

datatype ('*a*, '*b*) *splice-state* = *Left* '*a* '*b* | *Right* '*a* '*b* | *Left-only* '*a* | *Right-only* '*b*

fun *splice-raw* :: ('*a*, '*sg*) *raw-generator* ⇒ ('*a*, '*sh*) *raw-generator* ⇒ ('*a*, ('*sg*, '*sh*) *splice-state*) *raw-generator*
where
splice-raw g h (*Left-only sg*) = (*case g sg of*
Done ⇒ *Done* | *Skip sg'* ⇒ *Skip* (*Left-only sg'*) | *Yield a sg'* ⇒ *Yield a* (*Left-only sg'*))
| *splice-raw g h* (*Left sg sh*) = (*case g sg of*
Done ⇒ *Skip* (*Right-only sh*) | *Skip sg'* ⇒ *Skip* (*Left sg' sh*) | *Yield a sg'* ⇒ *Yield a* (*Right sg' sh*))
| *splice-raw g h* (*Right-only sh*) = (*case h sh of*
Done ⇒ *Done* | *Skip sh'* ⇒ *Skip* (*Right-only sh'*) | *Yield a sh'* ⇒ *Yield a* (*Right-only sh'*))
| *splice-raw g h* (*Right sg sh*) = (*case h sh of*
Done ⇒ *Skip* (*Left-only sg*) | *Skip sh'* ⇒ *Skip* (*Right sg sh'*) | *Yield a sh'* ⇒ *Yield a* (*Left sg sh'*))

lemma *terminates-splice-raw*:
assumes *g*: *terminates g* **and** *h*: *terminates h*
shows *terminates* (*splice-raw g h*)
proof (*rule terminatesI*)
fix *s*

```
  { fix sg
    from g have sg ∈ terminates-on g by (simp add: terminates-def)
    hence Left-only sg ∈ terminates-on (splice-raw g h)
      by induction(auto intro: terminates-on.intros)
  } moreover {
    fix sh
    from h have sh ∈ terminates-on h by (simp add: terminates-def)
    hence Right-only sh ∈ terminates-on (splice-raw g h)
      by induction(auto intro: terminates-on.intros)
  } moreover {
    fix sg sh
    from g have sg ∈ terminates-on g by (simp add: terminates-def)
    hence Left sg sh ∈ terminates-on (splice-raw g h)
    proof (induction sg arbitrary: sh)
      case (unfold sg a sg')
      from h have sh ∈ terminates-on h by (simp add: terminates-def)
      hence Right sg' sh ∈ terminates-on (splice-raw g h)
        by induction(auto intro: terminates-on.intros unfold.IH calculation)
      thus ?case using unfold.hyps by (auto intro: terminates-on.unfold)
    qed(auto intro: terminates-on.intros calculation)
  } moreover {
    fix sg sh
    from h have sh ∈ terminates-on h by (simp add: terminates-def)
    hence Right sg sh ∈ terminates-on (splice-raw g h)
      by(induction sh arbitrary: sg)(auto intro: terminates-on.intros calculation) }
  ultimately show s ∈ terminates-on (splice-raw g h) by(cases s)(simp-all)
qed

lift-definition splice-trans :: ('a, 'sg) generator ⇒ ('a, 'sh) generator ⇒ ('a, ('sg, 'sh)
splice-state) generator
is splice-raw by (rule terminates-splice-raw)

lemma unstream-splice-trans-Right-only: unstream (splice-trans g h) (Right-only sh) =
unstream h sh
proof (induction sh taking: h rule: unstream.induct)
  case (1 sh)
  then show ?case by (cases generator h sh)(simp-all add: splice-trans.rep-eq)
qed

lemma unstream-splice-trans-Left-only: unstream (splice-trans g h) (Left-only sg) = un-
stream g sg
proof (induction sg taking: g rule: unstream.induct)
  case (1 sg)
  then show ?case by (cases generator g sg)(simp-all add: splice-trans.rep-eq)
```

qed

lemma *unstream-splice-trans* [*stream-fusion*]:
unstream (*splice-trans g h*) (*Left sg sh*) = *splice* (*unstream g sg*) (*unstream h sh*)
proof (*induction sg arbitrary*: *sh taking*: *g rule*: *unstream.induct*)
case (*1 sg sh*)
then show *?case*
proof (*cases generator g sg*)
case *Done*
with *unstream-splice-trans-Right-only*[*of g h*]
show *?thesis* **by** (*simp add*: *splice-trans.rep-eq*)
next
case (*Skip sg'*)
then show *?thesis* **using** *1.IH*(*1*) **by** (*simp add*: *splice-trans.rep-eq*)
next
case (*Yield a sg'*)
note *IH* = *1.IH*(*2*)[*OF Yield*]

have *a* # (*unstream* (*splice-trans g h*) (*Right sg' sh*)) = *splice* (*unstream g sg*) (*unstream h sh*)
proof (*induction sh taking*: *h rule*: *unstream.induct*)
case (*1 sh*)
show *?case*
proof (*cases generator h sh*)
case *Done*
with *unstream-splice-trans-Left-only*[*of g h sg'*]
show *?thesis* **using** *Yield* **by** (*simp add*: *splice-trans.rep-eq*)
next
case (*Skip sh'*)
then show *?thesis* **using** *Yield 1.IH*(*1*) *1.prems* **by**(*simp add*: *splice-trans.rep-eq*)
next
case (*Yield b sh'*)
then show *?thesis* **using** *IH* ‹*generator g sg* = *Yield a sg'*›
by (*simp add*: *splice-trans.rep-eq*)
qed
qed
then show *?thesis* **using** *Yield* **by** (*simp add*: *splice-trans.rep-eq*)
qed
qed

2.5.13 *list-update*

fun *list-update-raw* :: ('*a*,'*s*) *raw-generator* ⇒ '*a* ⇒ ('*a*, *nat* × '*s*) *raw-generator*
where

```
  list-update-raw g b (n, s) = (case g s of
    Done ⇒ Done | Skip s' ⇒ Skip (n, s')
  | Yield a s' ⇒ if n = 0 then Yield a (0,s')
                 else if n = 1 then Yield b (0, s')
                 else Yield a (n − 1, s'))

lemma terminates-list-update-raw:
  assumes terminates g
  shows terminates (list-update-raw g b)
proof (rule terminatesI)
  fix st :: nat × 'a
  obtain n s where st = (n, s) by (cases st)
  from assms have s ∈ terminates-on g by (simp add: terminates-def)
  then show st ∈ terminates-on (list-update-raw g b) unfolding ‹st = (n, s)›
  proof (induction s arbitrary: n)
    case (unfold s a s' n)
    then show (n, s) ∈ terminates-on (list-update-raw g b)
      by(cases n = 0 ∨ n = 1)(auto intro: terminates-on.unfold)
  qed(simp-all add: terminates-on.stop terminates-on.pause)
qed

lift-definition list-update-trans :: ('a,'s) generator ⇒ 'a ⇒ ('a, nat × 's)  generator
is list-update-raw by (rule terminates-list-update-raw)

lemma unstream-lift-update-trans-None: unstream (list-update-trans g b) (0, s) = un-
stream g s
proof(induction s taking: g rule: unstream.induct)
  case (1 s)
  then show ?case by (cases generator g s)(simp-all add: list-update-trans.rep-eq)
qed

lemma unstream-list-update-trans [stream-fusion]:
  unstream (list-update-trans g b) (Suc n, s) = list-update (unstream g s) n b
proof(induction s arbitrary: n taking: g rule: unstream.induct)
  case (1 s)
  then show ?case
  proof (cases generator g s)
    case Done
    then show ?thesis by (simp add: list-update-trans.rep-eq)
  next
    case (Skip s')
    then show ?thesis using 1.IH(1) by (simp add: list-update-trans.rep-eq)
  next
    case (Yield a s')
```

```
    then show ?thesis using unstream-lift-update-trans-None[of g b s'] 1.IH(2)
      by (cases n)(simp-all add: list-update-trans.rep-eq)
  qed
qed
```

2.5.14 *removeAll*

```
definition removeAll-raw :: 'a ⇒ ('a, 's) raw-generator ⇒ ('a, 's) raw-generator
where
 removeAll-raw b g s = (case g s of
    Done ⇒ Done | Skip s' ⇒ Skip s' | Yield a s' ⇒ if a = b then Skip s' else Yield a s')

lemma terminates-removeAll-raw:
  assumes terminates g
  shows terminates (removeAll-raw b g)
proof (rule terminatesI)
  fix s
  from assms have s ∈ terminates-on g by (simp add: terminates-def)
  then show s ∈ terminates-on (removeAll-raw b g)
  proof(induction s)
    case (unfold s a s')
    then show ?case
      by(cases a = b)(auto intro: terminates-on.intros simp add: removeAll-raw-def)
  qed(auto intro: terminates-on.intros simp add: removeAll-raw-def)
qed

lift-definition removeAll-trans :: 'a ⇒ ('a, 's) generator ⇒ ('a, 's) generator
is removeAll-raw by (rule terminates-removeAll-raw)

lemma unstream-removeAll-trans [stream-fusion]:
  unstream (removeAll-trans b g) s = removeAll b (unstream g s)
proof (induction s taking: g rule: unstream.induct)
  case (1 s)
  then show ?case
  proof(cases generator g s)
    case (Yield a s')
    then show ?thesis using 1.IH(2)
      by(cases a = b)(simp-all add: removeAll-trans.rep-eq removeAll-raw-def)
  qed(auto simp add: removeAll-trans.rep-eq removeAll-raw-def)
qed
```

2.5.15 *remove1*

```
fun remove1-raw :: 'a ⇒ ('a, 's) raw-generator ⇒ ('a, bool × 's) raw-generator
```

where
remove1-raw x g (*b*, *s*) = (*case g s of*
Done ⇒ *Done* | *Skip s′* ⇒ *Skip* (*b*, *s′*)
| *Yield y s′* ⇒ *if b* ∧ *x* = *y then Skip* (*False*, *s′*) *else Yield y* (*b*, *s′*))

lemma *terminates-remove1-raw*:
assumes *terminates g*
shows *terminates* (*remove1-raw b g*)
proof (*rule terminatesI*)
fix *st* :: *bool* × *′a*
obtain *c s* **where** *st* = (*c*, *s*) **by** (*cases st*)
from *assms* **have** *s* ∈ *terminates-on g* **by** (*simp add*: *terminates-def*)
then show *st* ∈ *terminates-on* (*remove1-raw b g*) **unfolding** ‹*st* = (*c*, *s*)›
proof (*induction s arbitrary*: *c*)
case (*stop s*)
then show *?case* **by** (*cases c*)(*simp-all add*: *terminates-on.stop*)
next
case (*pause s s′*)
then show *?case* **by** (*cases c*)(*simp-all add*: *terminates-on.pause*)
next
case (*unfold s a s′*)
then show *?case*
by(*cases c*)(*cases a* = *b*, *auto intro*: *terminates-on.intros*)
qed
qed

lift-definition *remove1-trans* :: *′a* ⇒ (*′a*, *′s*) *generator* ⇒ (*′a*, *bool* × *′s*) *generator*
is *remove1-raw* **by** (*rule terminates-remove1-raw*)

lemma *unstream-remove1-trans-False*: *unstream* (*remove1-trans b g*) (*False*, *s*) = *unstream g s*
proof (*induction s taking*: *g rule*: *unstream.induct*)
case (*1 s*)
then show *?case* **by** (*cases generator g s*)(*simp-all add*: *remove1-trans.rep-eq*)
qed

lemma *unstream-remove1-trans* [*stream-fusion*]:
unstream (*remove1-trans b g*) (*True*, *s*) = *remove1 b* (*unstream g s*)
proof(*induction s taking*: *g rule*: *unstream.induct*)
case (*1 s*)
then show *?case*
proof (*cases generator g s*)
case (*Yield a s′*)
then show *?thesis*

using *Yield 1.IH*(*2*) *unstream-remove1-trans-False*[*of b g*]
by (*cases a* = *b*)(*simp-all add*: *remove1-trans.rep-eq*)
qed(*simp-all add*: *remove1-trans.rep-eq*)
qed

2.5.16 (#)

fun *Cons-raw* :: *'a* ⇒ (*'a*, *'s*) *raw-generator* ⇒ (*'a*, *bool* × *'s*) *raw-generator*
where
Cons-raw x g (*b*, *s*) = (*if b then Yield x* (*False*, *s*) *else case g s of*
Done ⇒ *Done* | *Skip s'* ⇒ *Skip* (*False*, *s'*) | *Yield y s'* ⇒ *Yield y* (*False*, *s'*))

lemma *terminates-Cons-raw*:
assumes *terminates g*
shows *terminates* (*Cons-raw x g*)
proof (*rule terminatesI*)
fix *st* :: *bool* × *'a*
obtain *b s* **where** *st* = (*b*, *s*) **by** (*cases st*)
from *assms* **have** *s* ∈ *terminates-on g* **by** (*simp add*: *terminates-def*)
hence (*False*, *s*) ∈ *terminates-on* (*Cons-raw x g*)
by(*induction s arbitrary*: *b*)(*auto intro*: *terminates-on.intros*)
then show *st* ∈ *terminates-on* (*Cons-raw x g*) **unfolding** ‹*st* = (*b*, *s*)›
by(*cases b*)(*auto intro*: *terminates-on.intros*)
qed

lift-definition *Cons-trans* :: *'a* ⇒ (*'a*, *'s*) *generator* ⇒ (*'a*, *bool* × *'s*) *generator*
is *Cons-raw* **by**(*rule terminates-Cons-raw*)

lemma *unstream-Cons-trans-False*: *unstream* (*Cons-trans x g*) (*False*, *s*) = *unstream g s*
proof(*induction s taking*: *g rule*: *unstream.induct*)
case (*1 s*)
then show *?case* **by**(*cases generator g s*)(*auto simp add*: *Cons-trans.rep-eq*)
qed

We do not declare *Cons-trans* as a transformer. Otherwise, literal lists would be transformed into streams which adds a significant overhead to the stream state.

lemma *unstream-Cons-trans*: *unstream* (*Cons-trans x g*) (*True*, *s*) = *x* # *unstream g s*
using *unstream-Cons-trans-False*[*of x g s*] **by**(*simp add*: *Cons-trans.rep-eq*)

2.5.17 *List.maps*

Stream version based on Coutts [1].

We restrict the function for generating the inner lists to terminating generators because the code generator does not directly supported nesting abstract datatypes in other types.

fun *maps-raw*
  :: ('*a* ⇒ ('*b*, '*sg*) *generator* × '*sg*) ⇒ ('*a*, '*s*) *raw-generator*
  ⇒ ('*b*, '*s* × (('*b*, '*sg*) *generator* × '*sg*) *option*) *raw-generator*
where
  maps-raw f g (*s*, *None*) = (*case g s of*
    Done ⇒ *Done* | *Skip s'* ⇒ *Skip* (*s'*, *None*) | *Yield x s'* ⇒ *Skip* (*s'*, *Some* (*f x*)))
| *maps-raw f g* (*s*, *Some* (*g''*, *s''*)) = (*case generator g'' s'' of*
    Done ⇒ *Skip* (*s*, *None*) | *Skip s'* ⇒ *Skip* (*s*, *Some* (*g''*, *s'*)) | *Yield x s'* ⇒ *Yield x* (*s*, *Some* (*g''*, *s'*)))

lemma *terminates-on-maps-raw-Some*:
  assumes (*s*, *None*) ∈ *terminates-on* (*maps-raw f g*)
  shows (*s*, *Some* (*g''*, *s''*)) ∈ *terminates-on* (*maps-raw f g*)
proof −
  from *generator*[*of g''*] **have** *s''* ∈ *terminates-on* (*generator g''*) **by** (*simp add*: *terminates-def*)
  thus *?thesis* **by**(*induction*)(*auto intro*: *terminates-on.intros assms*)
qed

lemma *terminates-maps-raw*:
  assumes *terminates g*
  shows *terminates* (*maps-raw f g*)
proof
  fix *st* :: '*a* × (('*c*, '*d*) *generator* × '*d*) *option*
  obtain *s mgs* **where** *st* = (*s*, *mgs*) **by**(*cases st*)
  from *assms* **have** *s* ∈ *terminates-on g* **by** (*simp add*: *terminates-def*)
  then show *st* ∈ *terminates-on* (*maps-raw f g*) **unfolding** ‹*st* = (*s*, *mgs*)›
    apply(*induction arbitrary*: *mgs*)
    apply(*case-tac* [!] *mgs*)
    apply(*auto intro*: *terminates-on.intros intro*!: *terminates-on-maps-raw-Some*)
    done
qed

lift-definition *maps-trans* :: ('*a* ⇒ ('*b*, '*sg*) *generator* × '*sg*) ⇒ ('*a*, '*s*) *generator*
  ⇒ ('*b*, '*s* × (('*b*, '*sg*) *generator* × '*sg*) *option*) *generator*
is *maps-raw* **by**(*rule terminates-maps-raw*)

lemma *unstream-maps-trans-Some*:
  unstream (*maps-trans f g*) (*s*, *Some* (*g''*, *s''*)) = *unstream g'' s''* @ *unstream* (*maps-trans f g*) (*s*, *None*)
proof(*induction s'' taking*: *g'' rule*: *unstream.induct*)

```
  case (1 s'')
  then show ?case by(cases generator g'' s'')(simp-all add: maps-trans.rep-eq)
qed

lemma unstream-maps-trans:
  unstream (maps-trans f g) (s, None) = List.maps (case-prod unstream ∘ f) (unstream
g s)
proof(induction s taking: g rule: unstream.induct)
  case (1 s)
  thus ?case
  proof(cases generator g s)
    case (Yield x s')
    with 1.IH(2)[OF this] show ?thesis
      using unstream-maps-trans-Some[of f g - fst (f x) snd (f x)]
      by(simp add: maps-trans.rep-eq maps-simps split-def)
  qed(simp-all add: maps-trans.rep-eq maps-simps)
qed
```

The rule *unstream-map-trans* is too complicated for fusion because of *split*, which does not arise naturally from stream fusion rules. Moreover, according to Farmer et al. [2], this fusion is too general for further optimisations because the generators of the inner list are generated by the outer generator and therefore compilers may think that is was not known statically.

Instead, they propose a weaker version using *flatten* below. (More precisely, Coutts already mentions this approach in his PhD thesis [1], but dismisses it because it requires a stronger rewriting engine than GHC has. But Isabelle's simplifier language is sufficiently powerful.

```
fun fix-step :: 'a ⇒ ('b, 's) step ⇒ ('b, 'a × 's) step
where
  fix-step a Done = Done
| fix-step a (Skip s) = Skip (a, s)
| fix-step a (Yield x s) = Yield x (a, s)

fun fix-gen-raw :: ('a ⇒ ('b, 's) raw-generator) ⇒ ('b, 'a × 's) raw-generator
where fix-gen-raw g (a, s) = fix-step a (g a s)

lemma terminates-fix-gen-raw:
  assumes ⋀x. terminates (g x)
  shows terminates (fix-gen-raw g)
proof
  fix st :: 'a × 'b
  obtain a s where st = (a, s) by(cases st)
  from assms[of a] have s ∈ terminates-on (g a) by (simp add: terminates-def)
  then show st ∈ terminates-on (fix-gen-raw g) unfolding ‹st = (a, s)›
```

```
    by(induction)(auto intro: terminates-on.intros)
qed

lift-definition fix-gen :: ('a ⇒ ('b, 's) generator) ⇒ ('b, 'a × 's) generator
is fix-gen-raw by(rule terminates-fix-gen-raw)

lemma unstream-fix-gen: unstream (fix-gen g) (a, s) = unstream (g a) s
proof(induction s taking: g a rule: unstream.induct)
  case (1 s)
  thus ?case by(cases generator (g a) s)(simp-all add: fix-gen.rep-eq)
qed

context
  fixes f :: ('a ⇒ 's')
  and g'' :: ('b, 's') raw-generator
  and g :: ('a, 's) raw-generator
begin

fun flatten-raw :: ('b, 's × 's' option) raw-generator
where
  flatten-raw (s, None) = (case g s of
     Done ⇒ Done | Skip s' ⇒ Skip (s', None) | Yield x s' ⇒ Skip (s', Some (f x)))
| flatten-raw (s, Some s'') = (case g'' s'' of
     Done ⇒ Skip (s, None) | Skip s' ⇒ Skip (s, Some s') | Yield x s' ⇒ Yield x (s,
Some s'))

lemma terminates-flatten-raw:
  assumes terminates g'' terminates g
  shows terminates flatten-raw
proof
  fix st :: 's × 's' option
  obtain s ms where st = (s, ms) by(cases st)
  { fix s s''
    assume s: (s, None) ∈ terminates-on flatten-raw
    from ‹terminates g''› have s'' ∈ terminates-on g'' by (simp add: terminates-def)
    hence (s, Some s'') ∈ terminates-on flatten-raw
      by(induction)(auto intro: terminates-on.intros s) }
  note Some = this
  from ‹terminates g› have s ∈ terminates-on g by (simp add: terminates-def)
  then show st ∈ terminates-on flatten-raw unfolding ‹st = (s, ms)›
    apply(induction arbitrary: ms)
    apply(case-tac [!] ms)
    apply(auto intro: terminates-on.intros intro!: Some)
    done
```

qed

end

lift-definition *flatten* :: ('*a* ⇒ '*s*') ⇒ ('*b*, '*s*') *generator* ⇒ ('*a*, '*s*) *generator* ⇒ ('*b*, '*s* × '*s*' *option*) *generator*
is *flatten-raw* **by**(*fact terminates-flatten-raw*)

lemma *unstream-flatten-Some*:
unstream (*flatten f g*'' *g*) (*s*, *Some s*') = *unstream g*'' *s*' @ *unstream* (*flatten f g*'' *g*) (*s*, *None*)
proof(*induction s*' *taking*: *g*'' *rule*: *unstream.induct*)
case (*1 s*')
thus *?case* **by**(*cases generator g*'' *s*')(*simp-all add*: *flatten.rep-eq*)
qed

HO rewrite equations can express the variable capture in the generator unlike GHC rules

lemma *unstream-flatten-fix-gen* [*stream-fusion*]:
unstream (*flatten* (λ*s*. (*s*, *f s*)) (*fix-gen g*'') *g*) (*s*, *None*) =
List.maps (λ*s*'. *unstream* (*g*'' *s*') (*f s*')) (*unstream g s*)
proof(*induction s taking*: *g rule*: *unstream.induct*)
case (*1 s*)
thus *?case*
proof(*cases generator g s*)
case (*Yield x s*')
with *1.IH*(*2*)[*OF this*] *unstream-flatten-Some*[*of* λ*s*. (*s*, *f s*) *fix-gen g*'' *g*]
show *?thesis*
by(*subst* (*1 3*) *unstream.simps*)(*simp add*: *flatten.rep-eq maps-simps unstream-fix-gen*)
qed(*simp-all add*: *flatten.rep-eq maps-simps*)
qed

Separate fusion rule when the inner generator does not depend on the elements of the outer stream.

lemma *unstream-flatten* [*stream-fusion*]:
unstream (*flatten f g*'' *g*) (*s*, *None*) = *List.maps* (λ*s*'. *unstream g*'' (*f s*')) (*unstream g s*)
proof(*induction s taking*: *g rule*: *unstream.induct*)
case (*1 s*)
thus *?case*
proof(*cases generator g s*)
case (*Yield x s*')
with *1.IH*(*2*)[*OF this*] **show** *?thesis*
using *unstream-flatten-Some*[*of f g*'' *g s*' *f x*]
by(*simp add*: *flatten.rep-eq maps-simps o-def*)

qed(*simp-all add*: *maps-simps flatten.rep-eq*)
qed

end

3 Stream fusion for coinductive lists

theory *Stream-Fusion-LList* **imports**
Stream-Fusion-List
Coinductive.Coinductive-List
begin

There are two choices of how many *Skip*s may occur consecutively.

- A generator for *'a llist* may return only finitely many *Skip*s before it has to decide on a *Done* or *Yield*. Then, we can define stream versions for all functions that can be defined by corecursion up-to. This in particular excludes *lfilter*. Moreover, we have to prove that every generator satisfies this restriction.
- A generator for *'a llist* may return infinitely many *Skip*s in a row. Then, the *lunstream* function suffers from the same difficulties as *lfilter* with definitions, but we can define it using the least fixpoint approach described in [4]. Consequently, we can only fuse transformers that are monotone and continuous with respect to the ccpo ordering. This in particular excludes *lappend*.

Here, we take the both approaches where we consider the first preferable to the second. Consequently, we define producers such that they produce generators of the first kind, if possible. There will be multiple equations for transformers and consumers that deal with all the different combinations for their parameter generators. Transformers should yield generators of the first kind whenever possible. Consumers can be defined using *lunstream* and refined with custom code equations, i.e., they can operate with infinitely many *Skip*s in a row. We just have to lift the fusion equation to the first kind, too.

type-synonym $('a, 's)$ *lgenerator* $= 's \Rightarrow ('a, 's)$ *step*

inductive-set *productive-on* :: $('a, 's)$ *lgenerator* $\Rightarrow 's$ *set*
for $g :: ('a, 's)$ *lgenerator*
where
Done: $g\ s = Done \Longrightarrow s \in productive\text{-}on\ g$
| *Skip*: $[\![\ g\ s = Skip\ s';\ s' \in productive\text{-}on\ g\ ]\!] \Longrightarrow s \in productive\text{-}on\ g$
| *Yield*: $g\ s = Yield\ x\ s' \Longrightarrow s \in productive\text{-}on\ g$

definition *productive* :: $('a, 's)$ *lgenerator* $\Rightarrow$ *bool*
where $productive\ g \longleftrightarrow productive\text{-}on\ g = UNIV$

lemma *productiveI* [*intro?*]:
$(\bigwedge s.\ s \in productive\text{-}on\ g) \Longrightarrow productive\ g$

by(*auto simp add*: *productive-def*)

lemma *productive-onI* [*dest?*]: *productive g* $\Longrightarrow$ *s* $\in$ *productive-on g*
by(*simp add*: *productive-def*)

A type of generators that eventually will yield something else than a skip.

typedef (*'a*, *'s*) *lgenerator'* = {*g* :: (*'a*, *'s*) *lgenerator*. *productive g*}
  morphisms *lgenerator Abs-lgenerator'*
proof
  show (*λ-*. *Done*) $\in$ *?lgenerator'* **by**(*auto intro*: *productive-on.intros productiveI*)
qed

setup-lifting *type-definition-lgenerator'*

3.1 Conversions to *'a llist*

3.1.1 Infinitely many consecutive *Skip*s

context fixes *g* :: (*'a*, *'s*) *lgenerator*
  notes [[*function-internals*]]
begin

partial-function (*llist*) *lunstream* :: *'s* $\Rightarrow$ *'a llist*
where
  lunstream s = (*case g s of*
    Done $\Rightarrow$ *LNil* | *Skip s'* $\Rightarrow$ *lunstream s'* | *Yield x s'* $\Rightarrow$ *LCons x* (*lunstream s'*))

declare *lunstream.simps*[*code*]

lemma *lunstream-simps*:
  g s = *Done* $\Longrightarrow$ *lunstream s* = *LNil*
  g s = *Skip s'* $\Longrightarrow$ *lunstream s* = *lunstream s'*
  g s = *Yield x s'* $\Longrightarrow$ *lunstream s* = *LCons x* (*lunstream s'*)
by(*simp-all add*: *lunstream.simps*)

lemma *lunstream-sels*:
  shows *lnull-lunstream*: *lnull* (*lunstream s*) $\longleftrightarrow$
  (*case g s of Done* $\Rightarrow$ *True* | *Skip s'* $\Rightarrow$ *lnull* (*lunstream s'*) | *Yield - -* $\Rightarrow$ *False*)
  and *lhd-lunstream*: *lhd* (*lunstream s*) =
  (*case g s of Skip s'* $\Rightarrow$ *lhd* (*lunstream s'*) | *Yield x -* $\Rightarrow$ *x*)
  and *ltl-lunstream*: *ltl* (*lunstream s*) =
  (*case g s of Done* $\Rightarrow$ *LNil* | *Skip s'* $\Rightarrow$ *ltl* (*lunstream s'*) | *Yield - s'* $\Rightarrow$ *lunstream s'*)
by(*simp-all add*: *lhd-def lunstream-simps split*: *step.split*)

end

3.1.2 Finitely many consecutive *Skip*s

lift-definition *lunstream'* :: ('*a*, '*s*) *lgenerator'* ⇒ '*s* ⇒ '*a llist*
is *lunstream* **.**

lemma *lunstream'-simps*:
lgenerator g s = *Done* ⟹ *lunstream' g s* = *LNil*
lgenerator g s = *Skip s'* ⟹ *lunstream' g s* = *lunstream' g s'*
lgenerator g s = *Yield x s'* ⟹ *lunstream' g s* = *LCons x* (*lunstream' g s'*)
by(*transfer*, *simp add*: *lunstream-simps*)+

lemma *lunstream'-sels*:
shows *lnull-lunstream'*: *lnull* (*lunstream' g s*) ⟷
(*case lgenerator g s of Done* ⇒ *True* | *Skip s'* ⇒ *lnull* (*lunstream' g s'*) | *Yield* - - ⇒ *False*)
and *lhd-lunstream'*: *lhd* (*lunstream' g s*) =
(*case lgenerator g s of Skip s'* ⇒ *lhd* (*lunstream' g s'*) | *Yield x* - ⇒ *x*)
and *ltl-lunstream'*: *ltl* (*lunstream' g s*) =
(*case lgenerator g s of Done* ⇒ *LNil* | *Skip s'* ⇒ *ltl* (*lunstream' g s'*) | *Yield* - *s'* ⇒ *lunstream' g s'*)
by(*transfer*, *simp add*: *lunstream-sels*)+

setup ‹*Context.theory-map* (*fold*
Stream-Fusion.add-unstream [@{*const-name lunstream*}, @{*const-name lunstream'*}])›

3.2 Producers

3.2.1 Conversion to streams

fun *lstream* :: ('*a*, '*a llist*) *lgenerator*
where
lstream LNil = *Done*
| *lstream* (*LCons x xs*) = *Yield x xs*

lemma *case-lstream-conv-case-llist*:
(*case lstream xs of Done* ⇒ *done* | *Skip xs'* ⇒ *skip xs'* | *Yield x xs'* ⇒ *yield x xs'*) =
(*case xs of LNil* ⇒ *done* | *LCons x xs'* ⇒ *yield x xs'*)
by(*simp split*: *llist.split*)

lemma *mcont2mcont-lunstream*[*THEN llist.mcont2mcont*, *simp*, *cont-intro*]:
shows *mcont-lunstream*: *mcont lSup lprefix lSup lprefix* (*lunstream lstream*)
by(*rule llist.fixp-preserves-mcont1*[*OF lunstream.mono lunstream-def*])(*simp add*: *case-lstream-conv-case-ll*

lemma *lunstream-lstream*: *lunstream lstream xs* = *xs*
by(*induction xs*)(*simp-all add*: *lunstream-simps*)

lift-definition *lstream'* :: (*'a*, *'a llist*) *lgenerator'*
is *lstream*
proof
  fix *s* :: *'a llist*
  show *s* ∈ *productive-on lstream* **by**(*cases s*)(*auto intro*: *productive-on.intros*)
qed

lemma *lunstream'-lstream*: *lunstream' lstream' xs* = *xs*
by(*transfer*)(*rule lunstream-lstream*)

3.2.2 *iterates*

definition *iterates-raw* :: (*'a* ⇒ *'a*) ⇒ (*'a*, *'a*) *lgenerator*
where *iterates-raw f s* = *Yield s* (*f s*)

lemma *lunstream-iterates-raw*: *lunstream* (*iterates-raw f*) *x* = *iterates f x*
by(*coinduction arbitrary*: *x*)(*auto simp add*: *iterates-raw-def lunstream-sels*)

lift-definition *iterates-prod* :: (*'a* ⇒ *'a*) ⇒ (*'a*, *'a*) *lgenerator'* **is** *iterates-raw*
by(*auto 4 3 intro*: *productiveI productive-on.intros simp add*: *iterates-raw-def*)

lemma *lunstream'-iterates-prod* [*stream-fusion*]: *lunstream'* (*iterates-prod f*) *x* = *iterates f x*
by *transfer*(*rule lunstream-iterates-raw*)

3.2.3 *unfold-llist*

definition *unfold-llist-raw* :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'a*) ⇒ (*'b*, *'a*) *lgenerator*
where
  unfold-llist-raw stop head tail s = (*if stop s then Done else Yield* (*head s*) (*tail s*))

lemma *lunstream-unfold-llist-raw*:
  lunstream (*unfold-llist-raw stop head tail*) *s* = *unfold-llist stop head tail s*
by(*coinduction arbitrary*: *s*)(*auto simp add*: *lunstream-sels unfold-llist-raw-def*)

lift-definition *unfold-llist-prod* :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'a*) ⇒ (*'b*, *'a*) *lgenerator'*
is *unfold-llist-raw*
proof(*rule productiveI*)
  fix *stop* **and** *head* :: *'a* ⇒ *'b* **and** *tail s*
  show *s* ∈ *productive-on* (*unfold-llist-raw stop head tail*)
    by(*cases stop s*)(*auto intro*: *productive-on.intros simp add*: *unfold-llist-raw-def*)
qed

lemma *lunstream'-unfold-llist-prod* [*stream-fusion*]:
lunstream' (*unfold-llist-prod stop head tail*) *s* = *unfold-llist stop head tail s*
by *transfer*(*rule lunstream-unfold-llist-raw*)

3.2.4 *inf-llist*

definition *inf-llist-raw* :: (*nat* ⇒ *'a*) ⇒ (*'a*, *nat*) *lgenerator*
where *inf-llist-raw f n* = *Yield* (*f n*) (*Suc n*)

lemma *lunstream-inf-llist-raw*: *lunstream* (*inf-llist-raw f*) *n* = *ldropn n* (*inf-llist f*)
by(*coinduction arbitrary*: *n*)(*auto simp add*: *lunstream-sels inf-llist-raw-def*)

lift-definition *inf-llist-prod* :: (*nat* ⇒ *'a*) ⇒ (*'a*, *nat*) *lgenerator'* **is** *inf-llist-raw*
by(*auto 4 3 intro*: *productiveI productive-on.intros simp add*: *inf-llist-raw-def*)

lemma *inf-llist-prod-fusion* [*stream-fusion*]:
lunstream' (*inf-llist-prod f*) *0* = *inf-llist f*
by *transfer*(*simp add*: *lunstream-inf-llist-raw*)

3.3 Consumers

3.3.1 *lhd*

context fixes *g* :: (*'a*, *'s*) *lgenerator* **begin**

definition *lhd-cons* :: *'s* ⇒ *'a*
where [*stream-fusion*]: *lhd-cons s* = *lhd* (*lunstream g s*)

lemma *lhd-cons-code*[*code*]:
lhd-cons s = (*case g s of Done* ⇒ *undefined* | *Skip s'* ⇒ *lhd-cons s'* | *Yield x* - ⇒ *x*)
by(*simp add*: *lhd-cons-def lunstream-simps lhd-def split*: *step.split*)

end

lemma *lhd-cons-fusion2* [*stream-fusion*]:
lhd-cons (*lgenerator g*) *s* = *lhd* (*lunstream' g s*)
by *transfer*(*rule lhd-cons-def*)

3.3.2 *llength*

context fixes *g* :: (*'a*, *'s*) *lgenerator* **begin**

definition *gen-llength-cons* :: *enat* ⇒ *'s* ⇒ *enat*
where *gen-llength-cons n s* = *n* + *llength* (*lunstream g s*)

lemma *gen-llength-cons-code* [*code*]:
gen-llength-cons n s = (*case g s of*
Done ⇒ *n* | *Skip s′* ⇒ *gen-llength-cons n s′* | *Yield - s′* ⇒ *gen-llength-cons* (*eSuc n*) *s′*)
by(*simp add*: *gen-llength-cons-def lunstream-simps iadd-Suc-right iadd-Suc split*: *step.split*)

lemma *gen-llength-cons-fusion* [*stream-fusion*]:
gen-llength-cons 0 s = *llength* (*lunstream g s*)
by(*simp add*: *gen-llength-cons-def*)

end

context fixes *g* :: (′*a*, ′*s*) *lgenerator′* **begin**

definition *gen-llength-cons′* :: *enat* ⇒ ′*s* ⇒ *enat*
where *gen-llength-cons′* = *gen-llength-cons* (*lgenerator g*)

lemma *gen-llength-cons′-code* [*code*]:
gen-llength-cons′ n s = (*case lgenerator g s of*
Done ⇒ *n* | *Skip s′* ⇒ *gen-llength-cons′ n s′* | *Yield - s′* ⇒ *gen-llength-cons′* (*eSuc n*) *s′*)
by(*simp add*: *gen-llength-cons′-def cong*: *step.case-cong*)(*rule gen-llength-cons-code*)

lemma *gen-llength-cons′-fusion* [*stream-fusion*]:
gen-llength-cons′ 0 s = *llength* (*lunstream′ g s*)
by(*simp add*: *gen-llength-cons′-def gen-llength-cons-fusion lunstream′.rep-eq*)

end

3.3.3 *lnull*

context fixes *g* :: (′*a*, ′*s*) *lgenerator* **begin**

definition *lnull-cons* :: ′*s* ⇒ *bool*
where [*stream-fusion*]: *lnull-cons s* ⟷ *lnull* (*lunstream g s*)

lemma *lnull-cons-code* [*code*]:
lnull-cons s ⟷ (*case g s of*
Done ⇒ *True* | *Skip s′* ⇒ *lnull-cons s′* | *Yield - -* ⇒ *False*)
by(*simp add*: *lnull-cons-def lunstream-simps split*: *step.split*)

end

context fixes $g :: ('a, 's)\ lgenerator'$ **begin**

definition *lnull-cons'* :: $'s \Rightarrow bool$
where $lnull\text{-}cons' = lnull\text{-}cons\ (lgenerator\ g)$

lemma *lnull-cons'-code* [*code*]:
$lnull\text{-}cons'\ s \longleftrightarrow (case\ lgenerator\ g\ s\ of$
$Done \Rightarrow True \mid Skip\ s' \Rightarrow lnull\text{-}cons'\ s' \mid Yield\ \text{-}\ \text{-} \Rightarrow False)$
by(*simp add*: *lnull-cons'-def cong*: *step.case-cong*)(*rule lnull-cons-code*)

lemma *lnull-cons'-fusion* [*stream-fusion*]:
$lnull\text{-}cons'\ s \longleftrightarrow lnull\ (lunstream'\ g\ s)$
by(*simp add*: *lnull-cons'-def lnull-cons-def lunstream'.rep-eq*)

end

3.3.4 *llist-all2*

context
fixes $g :: ('a, 'sg)\ lgenerator$
and $h :: ('b, 'sh)\ lgenerator$
and $P :: 'a \Rightarrow 'b \Rightarrow bool$
begin

definition *llist-all2-cons* :: $'sg \Rightarrow 'sh \Rightarrow bool$
where [*stream-fusion*]: $llist\text{-}all2\text{-}cons\ sg\ sh \longleftrightarrow llist\text{-}all2\ P\ (lunstream\ g\ sg)\ (lunstream\ h\ sh)$

definition *llist-all2-cons1* :: $'a \Rightarrow 'sg \Rightarrow 'sh \Rightarrow bool$
where $llist\text{-}all2\text{-}cons1\ x\ sg'\ sh = llist\text{-}all2\ P\ (LCons\ x\ (lunstream\ g\ sg'))\ (lunstream\ h\ sh)$

lemma *llist-all2-cons-code* [*code*]:
$llist\text{-}all2\text{-}cons\ sg\ sh =$
$(case\ g\ sg\ of$
$Done \Rightarrow lnull\text{-}cons\ h\ sh$
$\mid Skip\ sg' \Rightarrow llist\text{-}all2\text{-}cons\ sg'\ sh$
$\mid Yield\ a\ sg' \Rightarrow llist\text{-}all2\text{-}cons1\ a\ sg'\ sh)$
by(*simp split*: *step.split add*: *llist-all2-cons-def lnull-cons-def llist-all2-cons1-def lunstream-simps lnull-def*)

lemma *llist-all2-cons1-code* [*code*]:
$llist\text{-}all2\text{-}cons1\ x\ sg'\ sh =$
$(case\ h\ sh\ of$

Done ⇒ *False*
| *Skip sh′* ⇒ *llist-all2-cons1 x sg′ sh′*
| *Yield y sh′* ⇒ *P x y* ∧ *llist-all2-cons sg′ sh′*)
by(*simp split*: *step.split add*: *llist-all2-cons-def lnull-cons-def lnull-def llist-all2-cons1-def lunstream-simps*)

end

lemma *llist-all2-cons-fusion2* [*stream-fusion*]:
llist-all2-cons (*lgenerator g*) (*lgenerator h*) *P sg sh* ⟷ *llist-all2 P* (*lunstream′ g sg*) (*lunstream′ h sh*)
by *transfer*(*rule llist-all2-cons-def*)

lemma *llist-all2-cons-fusion3* [*stream-fusion*]:
llist-all2-cons g (*lgenerator h*) *P sg sh* ⟷ *llist-all2 P* (*lunstream g sg*) (*lunstream′ h sh*)
by *transfer*(*rule llist-all2-cons-def*)

lemma *llist-all2-cons-fusion4* [*stream-fusion*]:
llist-all2-cons (*lgenerator g*) *h P sg sh* ⟷ *llist-all2 P* (*lunstream′ g sg*) (*lunstream h sh*)
by *transfer*(*rule llist-all2-cons-def*)

3.3.5 *lnth*

context fixes *g* :: (′*a*, ′*s*) *lgenerator* **begin**

definition *lnth-cons* :: *nat* ⇒ ′*s* ⇒ ′*a*
where [*stream-fusion*]: *lnth-cons n s* = *lnth* (*lunstream g s*) *n*

lemma *lnth-cons-code* [*code*]:
lnth-cons n s = (*case g s of*
Done ⇒ *undefined n*
| *Skip s′* ⇒ *lnth-cons n s′*
| *Yield x s′* ⇒ (*if n* = *0 then x else lnth-cons* (*n* − *1*) *s′*))
by(*cases n*)(*simp-all add*: *lnth-cons-def lunstream-simps lnth-LNil split*: *step.split*)

end

lemma *lnth-cons-fusion2* [*stream-fusion*]:
lnth-cons (*lgenerator g*) *n s* = *lnth* (*lunstream′ g s*) *n*
by *transfer*(*rule lnth-cons-def*)

3.3.6 *lprefix*

context
  fixes $g :: ('a, 'sg)\ lgenerator$
  and $h :: ('a, 'sh)\ lgenerator$
begin

definition *lprefix-cons* $:: 'sg \Rightarrow 'sh \Rightarrow bool$
where [*stream-fusion*]: *lprefix-cons sg sh* $\longleftrightarrow$ *lprefix* (*lunstream g sg*) (*lunstream h sh*)

definition *lprefix-cons1* $:: 'a \Rightarrow 'sg \Rightarrow 'sh \Rightarrow bool$
where *lprefix-cons1 x sg' sh* $\longleftrightarrow$ *lprefix* (*LCons x* (*lunstream g sg'*)) (*lunstream h sh*)

lemma *lprefix-cons-code* [*code*]:
  lprefix-cons sg sh $\longleftrightarrow$ (*case g sg of*
    Done $\Rightarrow$ *True* | *Skip sg'* $\Rightarrow$ *lprefix-cons sg' sh* | *Yield x sg'* $\Rightarrow$ *lprefix-cons1 x sg' sh*)
by(*simp add*: *lprefix-cons-def lprefix-cons1-def lunstream-simps split*: *step.split*)

lemma *lprefix-cons1-code* [*code*]:
  lprefix-cons1 x sg' sh $\longleftrightarrow$ (*case h sh of*
    Done $\Rightarrow$ *False* | *Skip sh'* $\Rightarrow$ *lprefix-cons1 x sg' sh'*
  | *Yield y sh'* $\Rightarrow$ $x = y \wedge$ *lprefix-cons sg' sh'*)
by(*simp add*: *lprefix-cons-def lprefix-cons1-def lunstream-simps split*: *step.split*)

end

lemma *lprefix-cons-fusion2* [*stream-fusion*]:
  lprefix-cons (*lgenerator g*) (*lgenerator h*) *sg sh* $\longleftrightarrow$ *lprefix* (*lunstream' g sg*) (*lunstream' h sh*)
by *transfer*(*rule lprefix-cons-def*)

lemma *lprefix-cons-fusion3* [*stream-fusion*]:
  lprefix-cons g (*lgenerator h*) *sg sh* $\longleftrightarrow$ *lprefix* (*lunstream g sg*) (*lunstream' h sh*)
by *transfer*(*rule lprefix-cons-def*)

lemma *lprefix-cons-fusion4* [*stream-fusion*]:
  lprefix-cons (*lgenerator g*) *h sg sh* $\longleftrightarrow$ *lprefix* (*lunstream' g sg*) (*lunstream h sh*)
by *transfer*(*rule lprefix-cons-def*)

3.4 Transformers

3.4.1 *lmap*

definition *lmap-trans* $:: ('a \Rightarrow 'b) \Rightarrow ('a, 's)\ lgenerator \Rightarrow ('b, 's)\ lgenerator$
where *lmap-trans* $=$ *map-raw*

lemma *lunstream-lmap-trans* [*stream-fusion*]: **fixes** *f g s*
defines [*simp*]: $g' \equiv$ *lmap-trans f g*
shows *lunstream g' s* = *lmap f* (*lunstream g s*) (**is** *?lhs* = *?rhs*)
proof(*rule lprefix-antisym*)
show *lprefix ?lhs ?rhs*
proof(*induction g' arbitrary*: *s rule*: *lunstream.fixp-induct*)
case (*3 lunstream-g'*)
then show *?case*
by(*cases g s*)(*simp-all add*: *lmap-trans-def map-raw-def lunstream-simps*)
qed *simp-all*
next
note [*cont-intro*] = *ccpo.admissible-leI*[*OF llist-ccpo*]
show *lprefix ?rhs ?lhs*
proof(*induction g arbitrary*: *s rule*: *lunstream.fixp-induct*)
case (*3 lunstream-g*)
thus *?case* **by**(*cases g s*)(*simp-all add*: *lmap-trans-def map-raw-def lunstream-simps*)
qed *simp-all*
qed

lift-definition *lmap-trans'* :: ('*a* ⇒ '*b*) ⇒ ('*a*, '*s*) *lgenerator'* ⇒ ('*b*, '*s*) *lgenerator'*
is *lmap-trans*
proof
fix *f* :: '*a* ⇒ '*b* **and** *g* :: ('*a*, '*s*) *lgenerator* **and** *s*
assume *productive g*
hence *s* ∈ *productive-on g* **..**
thus *s* ∈ *productive-on* (*lmap-trans f g*)
by *induction*(*auto simp add*: *lmap-trans-def map-raw-def intro*: *productive-on.intros*)
qed

lemma *lunstream'-lmap-trans'* [*stream-fusion*]:
lunstream' (*lmap-trans' f g*) *s* = *lmap f* (*lunstream' g s*)
by *transfer*(*rule lunstream-lmap-trans*)

3.4.2 *ltake*

fun *ltake-trans* :: ('*a*, '*s*) *lgenerator* ⇒ ('*a*, (*enat* × '*s*)) *lgenerator*
where
ltake-trans g (*n*, *s*) =
(*if n* = *0 then Done else case g s of*
Done ⇒ *Done* | *Skip s'* ⇒ *Skip* (*n*, *s'*) | *Yield a s'* ⇒ *Yield a* (*epred n*, *s'*))

lemma *ltake-trans-fusion* [*stream-fusion*]:
fixes *g' g*

defines [*simp*]: $g' \equiv$ *ltake-trans g*
shows *lunstream g'* (*n*, *s*) = *ltake n* (*lunstream g s*) (**is** *?lhs* = *?rhs*)
proof(*rule lprefix-antisym*)
show *lprefix ?lhs ?rhs*
proof(*induction g' arbitrary*: *n s rule*: *lunstream.fixp-induct*)
case (*3 lunstream-g'*)
thus *?case*
by(*cases g s*)(*auto simp add*: *lunstream-simps neq-zero-conv-eSuc*)
qed *simp-all*
show *lprefix ?rhs ?lhs*
proof(*induction g arbitrary*: *s n rule*: *lunstream.fixp-induct*)
case (*3 lunstream-g*)
thus *?case* **by**(*cases g s n rule*: *step.exhaust*[*case-product enat-coexhaust*])(*auto simp add*: *lunstream-simps*)
qed *simp-all*
qed

lift-definition *ltake-trans'* :: ('*a*, '*s*) *lgenerator'* ⇒ ('*a*, (*enat* × '*s*)) *lgenerator'*
is *ltake-trans*
proof
fix *g* :: ('*a*, '*s*) *lgenerator* **and** *s* :: *enat* × '*s*
obtain *n sg* **where** *s*: *s* = (*n*, *sg*) **by**(*cases s*)
assume *productive g*
hence *sg* ∈ *productive-on g* **..**
then show *s* ∈ *productive-on* (*ltake-trans g*) **unfolding** ‹*s* = (*n*, *sg*)›
apply(*induction arbitrary*: *n*)
apply(*case-tac* [!] *n rule*: *enat-coexhaust*)
apply(*auto intro*: *productive-on.intros*)
done
qed

lemma *ltake-trans'-fusion* [*stream-fusion*]:
lunstream' (*ltake-trans' g*) (*n*, *s*) = *ltake n* (*lunstream' g s*)
by *transfer*(*rule ltake-trans-fusion*)

3.4.3 *ldropn*

abbreviation (*input*) *ldropn-trans* :: ('*b*, '*a*) *lgenerator* ⇒ ('*b*, *nat* × '*a*) *lgenerator*
where *ldropn-trans* ≡ *drop-raw*

lemma *ldropn-trans-fusion* [*stream-fusion*]:
fixes *g* **defines** [*simp*]: $g' \equiv$ *ldropn-trans g*
shows *lunstream g'* (*n*, *s*) = *ldropn n* (*lunstream g s*) (**is** *?lhs* = *?rhs*)
proof(*rule lprefix-antisym*)

```
  show lprefix ?lhs ?rhs
  proof(induction g' arbitrary: n s rule: lunstream.fixp-induct)
    case (3 lunstream-g')
    thus ?case
      by(cases g s n rule: step.exhaust[case-product nat.exhaust])
        (auto simp add: lunstream-simps elim: meta-allE[where x=0])
  qed simp-all
  note [cont-intro] = ccpo.admissible-leI[OF llist-ccpo]
  show lprefix ?rhs ?lhs
  proof(induction g arbitrary: n s rule: lunstream.fixp-induct)
    case (3 lunstream-g)
   thus ?case by(cases n)(auto split: step.split simp add: lunstream-simps elim: meta-allE[where
x=0])
  qed simp-all
qed

lift-definition ldropn-trans' :: ('a, 's) lgenerator' ⇒ ('a, nat × 's) lgenerator'
is ldropn-trans
proof
  fix g :: ('a, 's) lgenerator and ns :: nat × 's
  obtain n s where ns: ns = (n, s) by(cases ns)
  assume g: productive g
  show ns ∈ productive-on (ldropn-trans g) unfolding ns
  proof(induction n arbitrary: s)
    case 0
    from g have s ∈ productive-on g ..
    thus ?case by induction(auto intro: productive-on.intros)
  next
    case (Suc n)
    from g have s ∈ productive-on g ..
    thus ?case by induction(auto intro: productive-on.intros Suc.IH)
  qed
qed

lemma ldropn-trans'-fusion [stream-fusion]:
  lunstream' (ldropn-trans' g) (n, s) = ldropn n (lunstream' g s)
by transfer(rule ldropn-trans-fusion)
```

3.4.4 *ldrop*

```
fun ldrop-trans :: ('a, 's) lgenerator ⇒ ('a, enat × 's) lgenerator
where
  ldrop-trans g (n, s) = (case g s of
    Done ⇒ Done | Skip s' ⇒ Skip (n, s')
```

| *Yield x s′* ⇒ (*if n = 0 then Yield x* (*n*, *s′*) *else Skip* (*epred n*, *s′*)))

lemma *ldrop-trans-fusion* [*stream-fusion*]:
fixes *g g′* **defines** [*simp*]: *g′* ≡ *ldrop-trans g*
shows *lunstream g′* (*n*, *s*) = *ldrop n* (*lunstream g s*) (**is** *?lhs* = *?rhs*)
proof(*rule lprefix-antisym*)
show *lprefix ?lhs ?rhs*
by(*induction g′ arbitrary*: *n s rule*: *lunstream.fixp-induct*)
(*auto simp add*: *lunstream-simps neq-zero-conv-eSuc elim*: *meta-allE*[**where** *x=0*] *split*: *step.split*)
show *lprefix ?rhs ?lhs*
proof(*induction g arbitrary*: *n s rule*: *lunstream.fixp-induct*)
case (*3 lunstream-g*)
thus *?case*
by(*cases n rule*: *enat-coexhaust*)(*auto simp add*: *lunstream-simps split*: *step.split elim*: *meta-allE*[**where** *x=0*])
qed *simp-all*
qed

lemma *ldrop-trans-fusion2* [*stream-fusion*]:
lunstream (*ldrop-trans* (*lgenerator g*)) (*n*, *s*) = *ldrop n* (*lunstream′ g s*)
by *transfer* (*rule ldrop-trans-fusion*)

3.4.5 *ltakeWhile*

abbreviation (*input*) *ltakeWhile-trans* :: (*′a* ⇒ *bool*) ⇒ (*′a*, *′s*) *lgenerator* ⇒ (*′a*, *′s*) *lgenerator*
where *ltakeWhile-trans* ≡ *takeWhile-raw*

lemma *ltakeWhile-trans-fusion* [*stream-fusion*]:
fixes *P g g′* **defines** [*simp*]: *g′* ≡ *ltakeWhile-trans P g*
shows *lunstream g′ s* = *ltakeWhile P* (*lunstream g s*) (**is** *?lhs* = *?rhs*)
proof(*rule lprefix-antisym*)
show *lprefix ?lhs ?rhs*
by(*induction g′ arbitrary*: *s rule*: *lunstream.fixp-induct*)(*auto simp add*: *lunstream-simps takeWhile-raw-def split*: *step.split*)
show *lprefix ?rhs ?lhs*
by(*induction g arbitrary*: *s rule*: *lunstream.fixp-induct*)(*auto split*: *step.split simp add*: *lunstream-simps takeWhile-raw-def*)
qed

lift-definition *ltakeWhile-trans′* :: (*′a* ⇒ *bool*) ⇒ (*′a*, *′s*) *lgenerator′* ⇒ (*′a*, *′s*) *lgenerator′*
is *ltakeWhile-trans*

proof
fix P **and** $g :: ('a, 's)$ *lgenerator* **and** s
assume *productive g*
hence $s \in$ *productive-on g* **..**
thus $s \in$ *productive-on* (*ltakeWhile-trans P g*)
apply(*induction*)
apply(*case-tac* [*3*] *P x*)
apply(*auto intro*: *productive-on.intros simp add*: *takeWhile-raw-def*)
done
qed

lemma *ltakeWhile-trans'-fusion* [*stream-fusion*]:
lunstream' (*ltakeWhile-trans' P g*) *s* = *ltakeWhile P* (*lunstream' g s*)
by *transfer*(*rule ltakeWhile-trans-fusion*)

3.4.6 *ldropWhile*

abbreviation (*input*) *ldropWhile-trans* :: ('*a* ⇒ *bool*) ⇒ ('*a*, '*b*) *lgenerator* ⇒ ('*a*, *bool* × '*b*) *lgenerator*
where *ldropWhile-trans* ≡ *dropWhile-raw*

lemma *ldropWhile-trans-fusion* [*stream-fusion*]:
fixes *P g g'* **defines** [*simp*]: *g'* ≡ *ldropWhile-trans P g*
shows *lunstream g'* (*True*, *s*) = *ldropWhile P* (*lunstream g s*) (**is** *?lhs* = *?rhs*)
proof −
have *lprefix ?lhs ?rhs lprefix* (*lunstream g'* (*False*, *s*)) (*lunstream g s*)
by(*induction g' arbitrary*: *s rule*: *lunstream.fixp-induct*)(*simp-all add*: *lunstream-simps split*: *step.split*)
moreover have *lprefix ?rhs ?lhs lprefix* (*lunstream g s*) (*lunstream g'* (*False*, *s*))
by(*induction g arbitrary*: *s rule*: *lunstream.fixp-induct*)(*simp-all add*: *lunstream-simps split*: *step.split*)
ultimately show *?thesis* **by**(*blast intro*: *lprefix-antisym*)
qed

lemma *ldropWhile-trans-fusion2* [*stream-fusion*]:
lunstream (*ldropWhile-trans P* (*lgenerator g*)) (*True*, *s*) = *ldropWhile P* (*lunstream' g s*)
by *transfer*(*rule ldropWhile-trans-fusion*)

3.4.7 *lzip*

abbreviation (*input*) *lzip-trans* :: ('*a*, '*s1*) *lgenerator* ⇒ ('*b*, '*s2*) *lgenerator* ⇒ ('*a* × '*b*, '*s1* × '*s2* × '*a option*) *lgenerator*
where *lzip-trans* ≡ *zip-raw*

lemma *lzip-trans-fusion* [*stream-fusion*]:
fixes *g h gh* **defines** [*simp*]: *gh* ≡ *lzip-trans g h*
shows *lunstream gh* (*sg*, *sh*, *None*) = *lzip* (*lunstream g sg*) (*lunstream h sh*)
(**is** *?lhs* = *?rhs*)
proof −
have *lprefix ?lhs ?rhs*
and ⋀*x*. *lprefix* (*lunstream gh* (*sg*, *sh*, *Some x*)) (*lzip* (*LCons x* (*lunstream g sg*)) (*lunstream h sh*))
proof(*induction gh arbitrary*: *sg sh rule*: *lunstream.fixp-induct*)
case (*3 lunstream*)
{ case *1* **show** *?case* **using** *3*
by(*cases g sg*)(*simp-all add*: *lunstream-simps*) **}**
{ case *2* **show** *?case* **using** *3*
by(*cases h sh*)(*simp-all add*: *lunstream-simps*) **}**
qed *simp-all*
moreover
note [*cont-intro*] = *ccpo.admissible-leI*[*OF llist-ccpo*]
have *lprefix ?rhs ?lhs*
and ⋀*x*. *lprefix* (*lzip* (*LCons x* (*lunstream g sg*)) (*lunstream h sh*)) (*lunstream gh* (*sg*, *sh*, *Some x*))
proof(*induction g arbitrary*: *sg sh rule*: *lunstream.fixp-induct*)
case (*3 lunstream-g*)
note *IH* = *3.IH*
{ case *1* **show** *?case* **using** *3*
by(*cases g sg*)(*simp-all add*: *lunstream-simps fun-ord-def*) **}**
{ case *2* **show** *?case*
proof(*induction h arbitrary*: *sh sg x rule*: *lunstream.fixp-induct*)
case (*3 unstream-h*)
thus *?case*
proof(*cases h sh*)
case (*Yield y sh′*)
thus *?thesis* **using** *3.prems IH 3.hyps*
by(*cases g sg*)(*auto 4 3 simp add*: *lunstream-simps fun-ord-def intro*: *monotone-lzip2*[*THEN monotoneD*] *lprefix-trans*)
qed(*simp-all add*: *lunstream-simps*)
qed *simp-all* **}**
next
case *2* **case** *2*
show *?case*
by(*induction h arbitrary*: *sh rule*: *lunstream.fixp-induct*)(*simp-all add*: *lunstream-simps split*: *step.split*)
qed *simp-all*
ultimately show *?thesis* **by**(*blast intro*: *lprefix-antisym*)

qed

lemma *lzip-trans-fusion2* [*stream-fusion*]:
lunstream (*lzip-trans* (*lgenerator g*) *h*) (*sg*, *sh*, *None*) = *lzip* (*lunstream′ g sg*) (*lunstream h sh*)
by *transfer*(*rule lzip-trans-fusion*)

lemma *lzip-trans-fusion3* [*stream-fusion*]:
lunstream (*lzip-trans g* (*lgenerator h*)) (*sg*, *sh*, *None*) = *lzip* (*lunstream g sg*) (*lunstream′ h sh*)
by *transfer*(*rule lzip-trans-fusion*)

lift-definition *lzip-trans′* :: (*′a*, *′s1*) *lgenerator′* ⇒ (*′b*, *′s2*) *lgenerator′* ⇒ (*′a* × *′b*, *′s1* × *′s2* × *′a option*) *lgenerator′*
is *lzip-trans*
proof
fix *g* :: (*′a*, *′s1*) *lgenerator* **and** *h* :: (*′b*, *′s2*) *lgenerator* **and** *s* :: *′s1* × *′s2* × *′a option*
assume *productive g* **and** *productive h*
obtain *sg sh mx* **where** *s*: *s* = (*sg*, *sh*, *mx*) **by**(*cases s*)
{ fix *x sg*
from ‹*productive h*› **have** *sh* ∈ *productive-on h* **..**
hence (*sg*, *sh*, *Some x*) ∈ *productive-on* (*lzip-trans g h*)
by(*induction*)(*auto simp add*: *intro*: *productive-on.intros*) **}**
moreover
from ‹*productive g*› **have** *sg* ∈ *productive-on g* **..**
then have (*sg*, *sh*, *None*) ∈ *productive-on* (*lzip-trans g h*)
by *induction*(*auto intro*: *productive-on.intros calculation*)
ultimately show *s* ∈ *productive-on* (*lzip-trans g h*) **unfolding** *s*
by(*cases mx*) *auto*
qed

lemma *lzip-trans′-fusion* [*stream-fusion*]:
lunstream′ (*lzip-trans′ g h*) (*sg*, *sh*, *None*) = *lzip* (*lunstream′ g sg*) (*lunstream′ h sh*)
by *transfer*(*rule lzip-trans-fusion*)

3.4.8 *lappend*

lift-definition *lappend-trans* :: (*′a*, *′sg*) *lgenerator′* ⇒ (*′a*, *′sh*) *lgenerator* ⇒ *′sh* ⇒ (*′a*, *′sg* + *′sh*) *lgenerator*
is *append-raw* **.**

lemma *lunstream-append-raw*:
fixes *g h sh gh* **defines** [*simp*]: *gh* ≡ *append-raw g h sh*
assumes *productive g*

shows *lunstream gh* (*Inl sg*) = *lappend* (*lunstream g sg*) (*lunstream h sh*)
proof(*coinduction arbitrary*: *sg rule*: *llist.coinduct-strong*)
case (*Eq-llist sg*)
{ fix *sh′*
have *lprefix* (*lunstream gh* (*Inr sh′*)) (*lunstream h sh′*)
by(*induction gh arbitrary*: *sh′ rule*: *lunstream.fixp-induct*)(*simp-all add*: *lunstream-simps split*: *step.split*)
moreover have *lprefix* (*lunstream h sh′*) (*lunstream gh* (*Inr sh′*))
by(*induction h arbitrary*: *sh′ rule*: *lunstream.fixp-induct*)(*simp-all add*: *lunstream-simps split*: *step.split*)
ultimately have *lunstream gh* (*Inr sh′*) = *lunstream h sh′*
by(*blast intro*: *lprefix-antisym*) **}**
note *Inr* = *this*[*unfolded gh-def*]
from ‹*productive g*› **have** *sg*: *sg* ∈ *productive-on g* **..**
then show *?case* **by** *induction*(*auto simp add*: *lunstream-sels Inr*)
qed

lemma *lappend-trans-fusion* [*stream-fusion*]:
lunstream (*lappend-trans g h sh*) (*Inl sg*) = *lappend* (*lunstream′ g sg*) (*lunstream h sh*)
by *transfer*(*rule lunstream-append-raw*)

lift-definition *lappend-trans′* :: ('*a*, '*sg*) *lgenerator′* ⇒ ('*a*, '*sh*) *lgenerator′* ⇒ '*sh* ⇒ ('*a*, '*sg* + '*sh*) *lgenerator′*
is *append-raw*
proof
fix *g* :: ('*a*, '*sg*) *lgenerator* **and** *h* :: ('*a*, '*sh*) *lgenerator* **and** *sh s*
assume *productive g productive h*
{ fix *sh′*
from ‹*productive h*› **have** *sh′* ∈ *productive-on h* **..**
then have *Inr sh′* ∈ *productive-on* (*append-raw g h sh*)
by *induction* (*auto intro*: *productive-on.intros*)
} moreover {
fix *sg*
from ‹*productive g*› **have** *sg* ∈ *productive-on g* **..**
then have *Inl sg* ∈ *productive-on* (*append-raw g h sh*)
by *induction*(*auto intro*: *productive-on.intros calculation*) **}**
ultimately show *s* ∈ *productive-on* (*append-raw g h sh*) **by**(*cases s*) *auto*
qed

lemma *lappend-trans′-fusion* [*stream-fusion*]:
lunstream′ (*lappend-trans′ g h sh*) (*Inl sg*) = *lappend* (*lunstream′ g sg*) (*lunstream′ h sh*)
by *transfer*(*rule lunstream-append-raw*)

3.4.9 *lfilter*

definition *lfilter-trans* :: ('*a* ⇒ *bool*) ⇒ ('*a*, '*s*) *lgenerator* ⇒ ('*a*, '*s*) *lgenerator*
where *lfilter-trans* = *filter-raw*

lemma *lunstream-lfilter-trans* [*stream-fusion*]:
  fixes *P g g'* **defines** [*simp*]: *g'* ≡ *lfilter-trans P g*
  shows *lunstream g' s* = *lfilter P* (*lunstream g s*) (**is** *?lhs* = *?rhs*)
proof(*rule lprefix-antisym*)
  show *lprefix ?lhs ?rhs*
    by(*induction g' arbitrary*: *s rule*: *lunstream.fixp-induct*)
      (*simp-all add*: *lfilter-trans-def filter-raw-def lunstream-simps split*: *step.split*)
  show *lprefix ?rhs ?lhs*
  by(*induction g arbitrary*: *s rule*: *lunstream.fixp-induct*)
    (*simp-all add*: *lfilter-trans-def filter-raw-def lunstream-simps split*: *step.split*)
qed

lemma *lunstream-lfilter-trans2* [*stream-fusion*]:
  lunstream (*lfilter-trans P* (*lgenerator g*)) *s* = *lfilter P* (*lunstream' g s*)
by *transfer*(*rule lunstream-lfilter-trans*)

3.4.10 *llist-of*

lift-definition *llist-of-trans* :: ('*a*, '*s*) *generator* ⇒ ('*a*, '*s*) *lgenerator'*
is λ*x*. *x*
proof
  fix *g* :: ('*a*, '*s*) *raw-generator* **and** *s*
  assume *terminates g*
  hence *s* ∈ *terminates-on g* **by**(*simp add*: *terminates-def*)
  then show *s* ∈ *productive-on g*
    by(*induction*)(*auto intro*: *productive-on.intros*)
qed

lemma *lunstream-llist-of-trans* [*stream-fusion*]:
  lunstream' (*llist-of-trans g*) *s* = *llist-of* (*unstream g s*)
apply(*induction s taking*: *g rule*: *unstream.induct*)
apply(*rule llist.expand*)
apply(*auto intro*: *llist.expand simp add*: *llist-of-trans.rep-eq lunstream-sels lunstream'.rep-eq split*: *step.split*)
done

We cannot define a stream version of *list-of* because we would have to test for finiteness first and therefore traverse the list twice.

end

4 Examples and test cases for stream fusion

theory *Stream-Fusion-Examples* **imports** *Stream-Fusion-LList* **begin**

lemma fixes *rhs z*
  defines *rhs* ≡ *nth-cons* (*flatten* (λ*s'*. *s'*) (*upto-prod 17*) (*upto-prod z*)) (*2*, *None*) *8*
  shows *nth* (*List.maps* (λ*x*. *upto x 17*) (*upto 2 z*)) *8* = *rhs*
using [[*simproc add*: *stream-fusion*, *stream-fusion-trace*]]
apply(*simp del*: *id-apply*) — fuses
by(*unfold rhs-def*) *rule*

lemma fixes *rhs z*
  defines *rhs* ≡ *nth-cons* (*flatten* (λ*s*. (*s*, *1*)) (*fix-gen* (λ*x*. *upto-prod* (*id x*))) (*upto-prod z*)) (*2*, *None*) *8*
  shows *nth* (*List.maps* (λ*x*. *upto 1* (*id x*)) (*upto 2 z*)) *8* = *rhs*
using [[*simproc add*: *stream-fusion*, *stream-fusion-trace*]]
apply(*simp del*: *id-apply*) — fuses
by(*unfold rhs-def*) *rule*

lemma fixes *rhs n*
  defines *rhs* ≡ *List.maps* (λ*x*. [*Suc 0*..<*sum-list-cons* (*replicate-prod x*) *x*]) [*2*..<*n*]
  shows (*concat* (*map* (λ*x*. [*1*..<*sum-list* (*replicate x x*)]) [*2*..<*n*])) = *rhs*
using [[*simproc add*: *stream-fusion*, *stream-fusion-trace*]]
apply(*simp add*: *concat-map-maps*) — fuses partially
by(*unfold rhs-def*) *rule*

4.1 Micro-benchmarks from Farmer et al. [2]

definition *test-enum* :: *nat* ⇒ *nat* — *id* required to avoid eta contraction
where *test-enum n* = *foldl* (+) *0* (*List.maps* (λ*x*. *upt 1* (*id x*)) (*upt 1 n*))

definition *test-nested* :: *nat* ⇒ *nat*
where *test-nested n* = *foldl* (+) *0* (*List.maps* (λ*x*. *List.maps* (λ*y*. *upt y x*) (*upt 1 x*)) (*upt 1 n*))

definition *test-merge* :: *integer* ⇒ *nat*
where *test-merge n* = *foldl* (+) *0* (*List.maps* (λ*x*. *if 2 dvd x then upt 1 x else upt 2 x*) (*upt 1* (*nat-of-integer n*)))

This rule performs the merge operation from [2, §5.2] for *if*. In general, we would also need it for all case operators.

lemma *unstream-if* [*stream-fusion*]:
  unstream (*if b then g else g'*) (*if b then s else s'*) =
  (*if b then unstream g s else unstream g' s'*)

by *simp*

lemma *if-same* [*code-unfold*]: (*if b then x else x*) = *x*
by *simp*

code-thms *test-enum*
code-thms *test-nested*
code-thms *test-merge*

4.2 Test stream fusion in the code generator

definition *fuse-test* :: *integer*
where *fuse-test* =
integer-of-int (*lhd* (*lfilter* (λ*x*. *x* < *1*) (*lappend* (*lmap* (λ*x*. *x* + *1*) (*llist-of* (*map* (λ*x*. *if x* = *0 then undefined else x*) [−*3*..*5*]))) (*repeat 3*))))

ML-val ‹*val* ~*2* = @{*code fuse-test*}› — If this test fails with exception Fail, then the stream fusion simproc failed. This test exploits that stream fusion introduces laziness.

end

References

[1] D. Coutts. *Stream Fusion: Practical shortcut fusion for coinductive sequence types.* PhD thesis, University of Oxford, 2010.

[2] A. Farmer, C. Höner zu Siederdissen, and A. Gill. The HERMIT in the stream. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM 2014)*, pages 97–108. ACM, 2014.

[3] B. Huffman. Stream fusion. *Archive of Formal Proofs*, 2009. http://isa-afp.org/entries/Stream-Fusion.shtml, Formal proof development.

[4] A. Lochbihler and J. Hölzl. Recursive functions on lazy lists via domains and topologies. volume 8558 of *LNCS (LNAI)*, pages 341–357. Springer, 2014.